Sandboxing with Capsicum

PAWEL JAKUB DAWIDEK AND MARIUSZ ZABORSKI

Pawel Jakub Dawidek is a co-founder and CTO at Wheel Systems and a FreeBSD committer who lives and works in Warsaw, Poland. He is the author of various GEOM classes, including the disk-encryption class GELI; he implemented the Highly Available Storage (HAST) daemon for distributing audit trail files (auditdistd), and nowadays is mostly working on the Capsicum framework and the Casper daemon. pjd@freebsd.org

Mariusz Zaborski is currently working as a software developer at Wheel Systems and is a student at Warsaw University of Technology. He is a successful Google Summer of Code 2013 student. His work is mostly focused on Capsicum and the Casper daemon. Mariusz's relationship with FreeBSD is still young but very intensive. oshogbo@freebsd.org

Very few programmers have managed to successfully use the principle of least privilege, as found in OpenSSH, Postfix, and djbdns. Capsicum, introduced in 2010, adds a capability model designed to make it easier for programmers to reason about how to split a program into privileged and unprivileged portions. In this article, we describe the changes made in Capsicum since 2010, compare Capsicum to earlier sandboxing techniques, and look at the new Casperd, which makes it simpler to split programs.

Long ago, people started to recognize that security models proposed by the mainstream operating systems, including Windows, Mac OS X, and all kinds of UNIX-like systems, are simply naive: All you need to do is to write programs that have no bugs. That's indeed naive. Let's also state an obvious rule: The more code we write, the more bugs we introduce, some of which may jeopardize the security of our system. Once we accept this fact, where do we go? We could only develop very small programs, which are easy to audit, but this again would be a bit naive.

To reduce the size of the TCB (trusted computing base), the privilege separation model was introduced. This model splits the program into several independent components, moving all privileged tasks to a small privileged process, and shifting all the work requiring no privileges but that may be risky (like processing network packets) to a larger process that has no privileges. In the case of OpenSSH, the unprivileged process is responsible for parsing all network packets, handling compression, encryption, etc., and the privileged process is responsible for authenticating credentials extracted by the unprivileged process, starting the user's shell, and so on. Those two processes communicate over pipes. Designing the separation properly is very important. If the unprivileged process would have been responsible for authentication and would just pass the result to the privileged process, the whole model would be useless [5, 6].

Global Namespaces

An unprivileged process should be enclosed within some kind of process sandbox. One way to evaluate how good the sandbox is is to check how many global namespaces it is protecting. By global namespace, we are referring to a limited area within the operating system, these areas having some set of names that allow the unambiguous identification of an object [2]. An example of a process sandbox is a Linux kernel mechanism called *seccomp*. This mechanism allows you to limit a process to a state in which you can't do any other system calls than *exit, sigreturn, read,* and *write* [3]. It appears to be a very secure approach, but it is also very restrictive. For example, you can't, in any situation, open any new file or receive a new file descriptor. Other mechanisms of process sandboxing are Seatbelt (in Mac OS X) and Capsicum (in FreeBSD), which will be described later in this article. In Table 1, we present a full list of the global namespaces in the FreeBSD kernel. One namespace example is the file paths global namespace, which is nothing more than the list of files, symlinks, and directories in our computer.

Namespace	Description
Process ID (PID)	UNIX processes are identified by unique IDs. PIDs are returned by `fork` and used for signal delivery, debugging, monitoring, and status collection.
File paths	UNIX files exist in a global, hierarchical namespace, which is protected by discretionary and mandatory access control.
NFS file handles	The NFS client and server identify files and directories on the wire using a flat, global file handle namespace. They are also exposed to processes to support the lock manager daemon and optimize local file access.
File system ID	File system IDs supplement paths to mount points, and are used for forcible unmount when there is no valid path to the mount point.
Protocol address	Protocol families use socket addresses to name local and foreign endpoints. These exist in global namespaces, such as IPv4 addresses and ports, or the file system namespace for local domain sockets.
Sysctl MIB	The `sysctl` management interface uses numbered and named entries, used to get or set system information, such as process lists and tuning parameters.
System V IPC	System V IPC message queues, semaphores, and shared memory segments exist in a flat, global integer namespace.
POSIX IPC	POSIX defines similar semaphore, message queue, and shared memory APIs with an undefined namespace: On some systems, these are mapped into the file system; on others they are simply a flat, global namespace.
System clocks	UNIX systems provide multiple interfaces for querying and manipulating one or more system clocks or timers.
Jails	The management namespace for FreeBSD-based virtualized environments.
CPU sets	A global namespace for affinity policies assigned to processes and threads.
Routing tables	A global namespace with routing tables assigned to process.

Table 1: Global namespaces in the FreeBSD operating system kernel [4]

Table 1 was first published in [4]. Additionally, we would like to include "routing tables" to this list. In the FreeBSD operating system, per-process routing tables may be changed using the program setfib(1).

Security Hacks

In this section, we describe many of the "sandboxing techniques" (or as we prefer, "security hacks") that were used before process sandboxing, and show that creating an isolated environment wasn't easy. Programmers try to simulate sandboxes using portable functions like *setuid(2)*, *setrlimit(2)*, *chroot(2)*, etc. Most of these functions are part of the POSIX standard, so they should work on Linux and UNIX operating systems.

setuid(2), setgid(2), and setgroups(2)

It is obvious that unprivileged processes cannot run with root privileges, so they have to run as some other user. In the past, it was common to choose the "nobody" user, but if multiple independent programs reuse this one UID to drop privileges, it may become possible to jump from one program to another. We don't want that. This is why programs nowadays reserve their own unprivileged users, like the "sshd" user in the case of OpenSSH. There are many details you have to do correctly or this won't work properly:

- When changing your UID, don't forget to change your GID, too.
- When changing your GID, be sure to do it before changing your UID or it will fail.
- When changing your GID, be sure to remove all the other groups the process owner (root) belongs to, and do it before changing UID.
- Be sure to use *setgroups(2)*, *setgid(2)*, and *setuid(2)* system calls, or it may be possible to switch back to root.
- Be sure to verify these operations actually succeed! On some systems, in some conditions, it is not possible for the root user to change its UID, for example, and you'll be left running as root.
- Be sure to verify that your target operating system's *setuid(2)* and *setgid(2)* system calls modify real, effective, and saved user ID and group ID (or use *setresuid(2)*/*setresgid(2)* if available).
- Be sure not to modify effective user ID before calling *setuid(2)* or it won't change saved UID, and it will be possible to switch back to root.
- Functions that allow you to change UID, GID, and groups require root privileges.

In Listing 1, we have provided an example implementation of this method. It looks easy, doesn't it? However, there are many examples of people making some slip-up trying to use this technique. The most common mistakes with CVE examples are:

- CVE-2013-4559 for lighttpd—missing checks for *setuid(2)*, *setgid(2)*, and *setgroups(2)* failures
- CVE-2007-0536 for rMake—missing *setgroups(2)* call
- CVE-2000-0172 for mtr—*seteuid(2)* instead of *setuid(2)*

```
 #define VERIFY(expr)   do      { \
         if (!(expr))           \
                 abort();        \
} while (0)
uid_truid, euid, suid;
gid_trgid, egid, sgid;
gid_tgidset[1];
gidset[0] = pw->pw_gid;
if (setgroups(1, gidset) == -1)
        err(1, Ünable to set groups to gid");
if (setgid(pw->pw_gid) == -1)
        err(1, Ünable to set gid");
if (setuid(pw->pw_uid) == -1)
        err(1, Ünable to set uid");
VERIFY(getresuid(&ruid, &euid, &suid)    == 0);
VERIFY(ruid == pw->pw_uid);
VERIFY(euid == pw->pw_uid);
VERIFY(suid == pw->pw_uid);
VERIFY(getresgid(&rgid, &egid, &sgid     == 0);
VERIFY(rgid == pw->pw_gid);
VERIFY(egid == pw->pw_gid);
VERIFY(sgid == pw->pw_gid);
VERIFY(getgroups(0, NULL) == 1);
VERIFY(getgroups(1, gidset) == 1);
VERIFY(gidset[0] == pw->pw_gid);
```

Listing 1: Example code to change UID and GID in a secure fashion

Directory Restrictions

The method just described provides us with some security in the file path namespace, but our unprivileged process can still access various files on the system, can fill up file systems like */tmp/*, or perform network communications. To "fix" the file system problem, we can use the *chroot(2)* system call, which limits access to the file system tree.

Again, a few traps await us here:

- The *chroot(2)* system call is limited to the root user only, so we need to do it before changing our UID!
- Once our root directory is changed we have to *chdir(2)* to the new "/" because if the process working directory is outside of the new root directory, it will remain possible to access all the files!
- Be careful not to leave any directory descriptors open or the process will be able to escape from within our new root directory!

Code which implemented most of these rules is presented in Listing 2. We skipped over checking every open directory descriptor and checking every component for ownership; however, you should be aware that leaving any open directory descriptor is a big mistake.

```
/* Check for open directory descriptors */
/* Check for ownership of every component */
if (chroot(dir) != 0)
        err(1, "Unable to change root directory to \
            %s", dir);
if (chdir("/") != 0)
        err(1, "Unable to change directory
            to new root");
```

Listing 2: Code demonstrating correct use of the *chroot(2)* function

Some examples of common mistakes, with corresponding CVEs:

- CVE-2008-5110, CVE-2011-4099—missing chdir("/") after *chroot(2)*
- CVE-2005-4532—chroot directory writable by user

P_SUGID

After changing our directory using *chroot(2)* and dropping privileges using *setuid(2)*, we are no longer running as root, but all our sandboxes run as the same UNPRIV_USER user, which is not good. For example, OpenSSH's sandbox is using the single *sshd* user to handle sessions from every user that is logging in, including root. Now if we break into such a sandbox we will be running as *sshd* user and can mess with other sandboxes, handling other SSH sessions. What exactly can we do? If we could use *ptrace(2)* to attach to a sandbox that handles root's session, then we could just modify this sandbox memory and break into root's SSH session. This possibility alone would make privilege separation useless. Fortunately, this is not possible. Because we were running as root and then dropped our privileges using *setuid(2)*, the kernel tagged our process with the *P_SUGID* flag. On FreeBSD, this prevents another process with the same user ID from being able to debug us. It also means that only some signals may be delivered to such a process, but those signals include *SIGUSR1*, *SIGUSR2*, *SIGHUP*, *SIGALRM*, etc., so it is still not without risk.

As we mentioned in the introduction to this section, most functions presented here are part of the POSIX standard and *should* work on most Linux and UNIX operating systems. Unfortunately, it is not all roses. For example, in 2005, Tavis Ormandy found out that the *setuid(2)* function does not set the *P_SUGID* flag in the NetBSD operating system [9]. So before sandboxing your process using all those techniques, be sure to check that they work properly on your destination operating system.

Very Restrictive Environment

The next thing we shall try to do is to prevent network connections. If an attacker can break into our program, they could, for example, run a spam-sending botnet. One way to prevent network connections is to set the limit on open file descriptors to zero, which will prevent the opening of any new file descriptors and raising the limit back.

Namespace	setuid(1)	chroot(2)	P_SUGID	setrlimit(2)	cap_enter(2)
Process IDs	Unprotected	Unprotected	Partial	Unprotected	Protected
File paths	Partial	Protected	Unprotected	Partial	Protected
NFS file handle	Protected	Unprotected	Unprotected	Unprotected	Protected
Filesystem IDs	Protected	Unprotected	Unprotected	Unprotected	Protected
Sysctl MIB	Partial	Unprotected	Partial	Unprotected	Protected
System V IPC	Unprotected	Unprotected	Unprotected	Unprotected	Protected
POSIX IPC	Partial	Unprotected	Unprotected	Protected	Protected
System clocks	Protected	Unprotected	Unprotected	Unprotected	Protected
Jails	Partial	Unprotected	Unprotected	Unprotected	Protected
CPU sets	Unprotected	Unprotected	Unprotected	Unprotected	Protected
Protocol address	Unprotected	Partial	Unprotected	Protected	Protected
Routing tables	Unprotected	Unprotected	Unprotected	Unprotected	Protected

Table 2: Showing which global namespaces are protected by different sandboxing techniques. Partial means the namespace is protected to some extent.

If we are limiting the number of file descriptors, we could also limit file size and disable forking. If we set the file size limit to zero, a process may not create any new files. Disabling forking will prevent any kind of DDoS attacks that involve running a lot of child processes.

Listing 3 shows example code that sets all of these restrictions.

```
structrlimitrl;
rl.rlim_cur = rl.rlim_max = 0;
if (setrlimit(RLIMIT_NOFILE, &rl) != 0)
   err(1, "Unable to limit file descriptors");
if (setrlimit(RLIMIT_FSIZE, &rl) != 0)
   err(1, "Unable to limit file size");
if (setrlimit(RLIMIT_NPROC, &rl) != 0)
   err(1, "Unable to disallow forking");
```

Listing 3: Example code to create a very restricted environment

This method is used, as far as the authors know, only in OpenSSH. These limits are very restrictive. The process may not receive any new file descriptors, duplicate any descriptors, or open any new files in any situation.

Summary of Security Hacks

These four methods are the most interesting methods to sandbox applications using standard functions. In Table 2, we present information on which method protects which namespace.

While analyzing the Table 2, please keep in mind that using *setrlimit(2)* technique imposes significant restrictions on the programmer and, in the common case, makes *setrlimit(2)* very impractical or even impossible to use.

As you can see, using those techniques leaves a lot of space for mistakes, without even covering all global namespaces. These methods also leave a lot of gaps in global namespaces that they should protect.

Capsicum

Capsicum is a lightweight OS capability and sandbox framework [7]. In FreeBSD, we can divide the architecture of our process sandbox system into two modules:

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

By "tight sandboxing" we understand that after calling the cap_enter(2) function, the FreeBSD kernel will disallow access to any global namespaces. The kernel will still allow access to any local namespaces, so we can continue to use any references to any part of the global namespace. For example, in the file path namespace you can open a directory (e.g., using the opendir(2) function), and after entering the sandbox you can still open any file within that directory (e.g., using the openat(2) function).

The second part of Capsicum consists of capability rights, which allow us to limit even more local namespaces. We have a lot of flexibility in setting capability rights, which we can limit to read-only, write-only, or append-only. Many limits are also namespace specific. For example, three file-specific rights are:

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

We also have socket-specific rights, for example:

- CAP_ACCEPT accepts connection on socket (*accept(2)*).
- CAP_BINDAT assigns a local protocol address to a socket (*bindat(2)*).

In the FreeBSD operating system, we have defined around 77 capabilities. A full list of the Capsicum capability rights can be found in the FreeBSD rights(4) manual page.

Implementation of Capability Rights

In FreeBSD, file descriptors are a carrier of capability rights. In a previous article about Capsicum [1], the authors wrote that we wrap a regular file descriptor structure in a special structure that holds information about rights. That has been changed twice since then. First, they were changed to remove the wrapper structure and add a variable to the filedescent structure to describe capability rights. The second modification was to change the type of the variable. Initially, rights were represented by the *uint64_t* type, allowing 64 rights to be defined. It turned out that the maximnt number of rights was too small and the *uint64_t* type was changed to a special structure that allows us to define up to 285 rights (and even more if needed with more involved changes).

This new structure is presented in Listing 4. The top two bits in the first element of the *cr_rights* array contain total number of elements in the array plus two. This means if those two bits are equal to 0, we have two array elements. The top two bits in all remaining array elements should be 0. The next five bits in all array elements contain an array index. Only one bit is used and bit position in this five-bit range defines the array index. This means there can be at most five array elements in the future. Using only one bit for array index helps to discover ORing rights from different array elements.

```
#define CAP_RIGHTS_VERSION_00   0
/*
 * #define CAP_RIGHTS_VERSION_01   1
 * #define CAP_RIGHTS_VERSION_02   2
 * #define CAP_RIGHTS_VERSION_03   3
 *
/
#define CAP_RIGHTS_VERSION   CAP_RIGHTS_VERSION_00

struct cap_rights {
        uint64_t cr_rights[CAP_RIGHTS_VERSION + 2]; };
typedefstructcap_rightscap_rights_t;
```

Listing 4: Current structure that defines Capsicum rights

Changing the type of the *cap_rights* structure also forces us to change the interface of the *cap_rights_limit(2)* function. In previous implementations to manage rights, we could simply use logic instructions (e.g., and, or), but now this is no longer possible. New interfaces are presented in Listing 5.

```
 /* Interfaces. */
cap_rights_t *cap_rights_init(cap_rights_t *rights, ...);
void cap_rights_set(cap_rights_t *rights, ...);
void cap_rights_clear(cap_rights_t *rights, ...);
bool cap_rights_is_set(constcap_rights_t *rights, ...);
```

Listing 5: New interfaces for managing capability rights

These functions replace the previous logic instructions. First, we need to initialize a *cap_rights_t* structure using *cap_rights_init(2)* function. Then we may add new rights using *cap_rights_set(2)*. Once we finish all the required operation settings, we can use *cap_rights_limit(2)* function to limit a file descriptor. All of these steps are presented in Listing 6.

```
intfd;
cap_rights_t rights;

cap_rights_init(&rights, CAP_READ, CAP_WRITE, CAP_FSTAT);
cap_rights_set(&rights, CAP_FCHMOD);

/* Limit descriptor */
cap_rights_limit(fd, &rights);
```

Listing 6: Example of new interface usage

Status of the Project

Capsicum was first introduced in FreeBSD 9.0 and from then was very quickly developed. Currently, there is ongoing work to port Capsicum sandbox to Linux, OpenBSD, and DragonFlyBSD [7]. A growing list of programs in the FreeBSD operating system now use Capsicum:

- auditdistd(8)
- dhclient(8)
- hastd(8)
- hastctl(8)
- kdump(1)
- rwho(1)
- rwhod(8)
- ping(8)
- sshd(8)
- tcpdump(8)
- uniq(1)

An up-to-date list can be found on the Cambridge Web site about Capsicum in FreeBSD [8].

Casper Daemon

Even though Capsicum gives us more flexibility than other methods, in some cases this is still not enough. Consider the situation in which you need to open a lot of different directories (for example, when sandboxing the *grep(1)* program), or the case where you need to open some Internet connection, but before entering the sandbox you don't know what kind of connection this will be.

The Capsicum framework resolves this problem using a privilege separation model. Before entering the sandbox you can spawn a new process which will have more access to global namespaces or even may not be sandboxed at all. The privileged process performs some operation like opening files or Internet connections and passes the file descriptor to the unprivileged process using UNIX domain sockets. The unprivileged process performs all other actions. The *rwhod(8)* utility is an example of a program that is sandboxed using this method.

This method works pretty well, but there is a lot of code that would need to be rewritten multiple times for different programs. To solve this problem, the Casper daemon was introduced.

Daemon Architecture

We can separate the Casper daemon into two parts: the Casper daemon itself *(casperd(8))* and Casper services.

The Casper daemon is a global program in an operating system that waits for connections from other process. We can establish a connection with the daemon using the *cap_init(2)* function. The Casper daemon automatically spawns a second process called the "zygote." The zygote is a lightweight process that closes all additional descriptors and uses minimal memory. When a process establishes a connection with the daemon, the process sends information about which services it will require. Casper receives that information and clones the zygote process, and after this operation, one zygote is transformed (using *execv(2)* function) into a service. The process shown in Figure 1 demonstrates these steps.

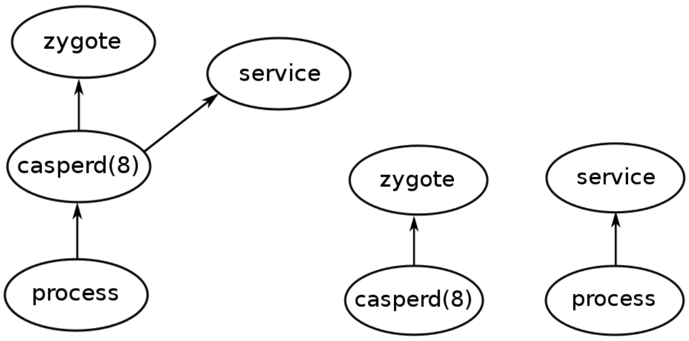

Figure 1: Life cycle of zygote in Casper daemon. On left side, the Casper daemon has spawned a zygote; on the right side, the zygote has been attached to the process-requesting service.

Casper services are specially written programs that have specific tasks. 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.

All of these services provide equivalent APIs to the function that they replace.

References

[1] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, Kris Kennaway, "Introducing Capsicum: Practical Capabilities for UNIX," *;login:*, vol. 35, no. 6 (December 2010): https://www.usenix.org/publications/login/december-2010-volume-35-number-6/introducing-capsicum-practical-capabilities-unix.

[2] "Namespace," Wikipedia, accessed September 11, 2014: http://en.wikipedia.org/wiki/Namespace.

[3] Google Seccomp Sandbox for Linux, 2014: https://code.google.com/p/seccompsandbox/wiki/overview.

[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] Niels Provos, Markus Friedl, Peter Honeyman, "Preventing Privilege Escalation": http://niels.xtdnet.nl/papers/privsep.pdf.

[6] Niels Provos, Privilege Separated OpenSSH: http://www.citi.umich.edu/u/provos/ssh/privsep.html.

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

[8] Robert Watson, Cambridge Computer Laboratory Web page—Capsicum FreeBSD, 2014: https://www.cl.cam.ac.uk/research/security/capsicum/freebsd.html.

[9] Tavis Ormandy, NetBSD Local PTrace Privilege Escalation Vulnerability, CVE-2005-4741: http://www.securityfocus.com/bid/15290/info.