SEE TEXT ONLY

WHATS GOING ON WITH CloudABI?

BY ED SCHOUTEN

The September/October 2015 issue of *FreeBSD Journal* featured an article on CloudABI, an open-source project I started working on earlier that same year. As a fair amount of time has passed since the article appeared, let's take a look at some of the developments that have taken place in the meantime and what is being planned next. But first, a recap of what CloudABI is and what using it looks like.

CLOUDABI IN THEORY

CloudABI is a runtime for which you can develop POSIX-like software. CloudABI is different from FreeBSD's native runtime in that it enforces programs to use dependency injection. Dependency injection is a technique that is normally used in the field of object-oriented programming to make components (classes) of a program easier to test and reuse. Instead of designing classes that attempt to communicate with the outside world directly, you design them so that communication channels are expressed as other objects on which the class may perform operations (dependencies).

A good example of dependency injection is a web server class that depends on a separate socket object being provided to its constructor. Such a class can be unit tested very easily by passing in a mock socket object that generates a fictive HTTP request and validates the web server's response. This class is also reusable, as support for different network protocols (IPv4 vs. IPv6, TCP vs. SCTP), cryptography (TLS), rate limiting, traffic shaping, etc., can all be added as separate helper classes. The implementation of the web server class itself can remain unchanged. A web server class that directly creates its own network socket through the `socket(2)`

system call wouldn't allow for this.

The goal behind CloudABI is to introduce dependency injection at a higher level. Instead of applying it to classes, we want to enforce entire programs to have all of their dependencies injected explicitly on startup. In our case, dependencies are represented by file descriptors. Programs can no longer open files using absolute pathnames. Files are only accessible by injecting a file descriptor corresponding to either the file itself or one of its parent directories. Programs can also no longer bind to arbitrary TCP ports. They can be injected with pre-bound sockets.

What is nice about this model is that it reduces the need for jails, containers, and virtual machines. These technologies are often used to overcome limitations related to the inability to run multiple configurations/versions of the same software alongside, due to a lack of isolation provided by UNIX-like systems by default. With CloudABI, it is possible to obtain such isolation by simply injecting every instance of a program with different sets of files, directories, and sockets.

It becomes very easy for an operating system to sandbox processes following this approach. By requiring all dependencies to be injected, the operating system can simply deny access to everything else. There is no need to maintain security policies separately, as is the case with frameworks like Linux's SELinux and AppArmor. If an attacker manages to take over control of a CloudABI-based process, he/she will effectively only be able to access the resources the process needs to interact with by design.

CloudABI has been influenced a lot by Capsicum, FreeBSD's capability-based security framework. CloudABI differs from Capsicum in that Capsicum still allows you to run startup code without having sandboxing enabled. The process has to switch over to "capabilities mode" manually, using `cap_enter(2)`. CloudABI executables are already sandboxed by the time the first instruction of the program gets executed.

The advantage of Capsicum's model is that it makes it possible to integrate sandboxing into conventional UNIX programs, which are typically not designed to have all of their dependencies injected. The advantage of CloudABI's model is that it allows us to remove all APIs that are incompatible with sandboxing. This reduces the effort needed to port software tremendously, as parts that need to be modified to work with sandboxing now trigger compiler errors, as opposed to runtime errors that may be hard to trigger, let alone debug.

A side effect of removing all of these sandboxing-incompatible APIs is that it makes CloudABI so compact that it can be implemented by other operating systems relatively easily. This means that you can use CloudABI to build a single executable that runs on multiple platforms without recompilation.

CLOUDABI IN PRACTICE

To demonstrate what it looks like to use CloudABI in practice, let's take a look at a tiny web server for CloudABI that is capable of returning a fixed HTML response back to the browser.

```
#include <sys/socket.h>
#include <argdata.h>
#include <program.h>
#include <string.h>
#include <unistd.h>

void program_main(const argdata_t *ad) {
   // Extract socket and message from config.
   int sockfd = -1;
   const char *message = "";
   {
       argdata_map_iterator_t it;
       argdata_map_iterate(ad, &it);
       const argdata_t *key, *value;
       while (argdata_map_next(&it, &key, &value)) {
            const char *keystr;
            if (argdata_get_str_c(key, &keystr) != 0)
               continue;
            if (strcmp(keystr, "http_socket") == 0)
               argdata_get_fd(value, &sockfd);
            else if (strcmp(keystr, "html_message") == 0)
               argdata_get_str_c(value, &message);
       }
   }

   // Handle incoming requests.
   // TODO: Actually process HTTP requests.
   // TODO: Use concurrency.
   for (;;) {
       int connfd = accept(sockfd, NULL, NULL);
       dprintf(connfd,
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: text/html\r\n"
            "Content-Length: %zu\r\n\r\n"
            "%s", strlen(message), message);
       close(connfd);
   }
}
```

What you may notice immediately is that CloudABI programs get started through a function called `program_main()`, as opposed to using C's standard `main()` function. The `program_main()` function does away with C's string command-line arguments and replaces it with a YAML/JSON-like tree structure called Argdata. In addition to storing values like booleans, integers, and strings, Argdata can have file descriptors attached to it. This is the mechanism that is used to inject dependencies on start-up. This web server expects the Argdata to be a map (dictionary), containing both a socket for accepting incoming requests (`http_socket`) and a HTML response string (`html_message`).

The following shell commands show how this web server can be built and executed. The web server can be compiled using a cross compiler provided by the `devel/cloudabi-toolchain` port. Once built, it can be started with `cloudabi-run`, which is provided by the `sysutils/cloudabi-utils` port. The `cloudabi-run` utility reads a YAML file from `stdin` and converts it to an Argdata tree, which is passed on to `program_main()`. The YAML file may contain tags like `!fd`, `!file`, and `!socket`. These tags are directives for `cloudabi-run` to insert file descriptors at those points in the Argdata tree. Only file descriptors referenced by the Argdata end up in the CloudABI process.

```
$ x86_64-unknown-cloudabi-cc -o webserver webserver.c
$ cat webserver.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
http_socket: !socket
    type: stream
    bind: 0.0.0.0:8080
html_message: <marquee>Hello, world!</marquee>
$ cloudabi-run webserver < webserver.yaml &
$ curl http://localhost:8080/
<marquee>Hello, world!</marquee>
```

This example shows that CloudABI can be used to build strongly sandboxed applications in an intuitive way. With the configuration passed to `cloudabi-run`, this web server is isolated from the rest of the system completely, with the exception of the HTTP socket on which it may accept incoming connections. By using Argdata, we can also omit a lot of boilerplate code from our web server, like configuration file parsing and socket creation. All of this functionality is implemented by `cloudabi-run` once and can be reused universally.

HARDWARE ARCHITECTURES

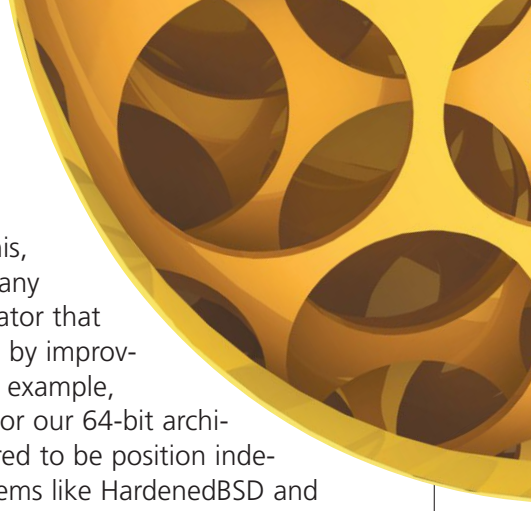
When CloudABI was released in 2015, we only provided support for creating executables for x86-64. As I believe CloudABI is a very useful tool for sandboxing software on embedded systems and appliances as well, we ported CloudABI to also work nicely on ARM64 around the same time the previous article on CloudABI was published. In August 2016, we ported CloudABI to the 32-bit equivalents of these architectures (i686 and ARMv6).

An interesting aspect of porting over CloudABI to these systems was to obtain a usable toolchain. When CloudABI was available only for x86-64, we already used Clang as our C/C++ compiler. Clang is nice in the sense that a single installation can be used to target multiple architectures very easily. It can automatically infer which architecture to use by inspecting `argv[0]` on startup. This meant that we only needed to extend the existing `devel/cloudabi-toolchain` port to install additional symbolic links pointing to Clang for every architecture that we support.

At the same time, we still made use of GNU Binutils to link our executables. Binutils has the disadvantage that an installation can only be used to target a single hardware architecture. Even worse, the Binutils codebase always requires a large number of modifications for every pair of operating system and hardware architecture it should support.

At around the time we started working on supporting more architectures, the LLVM project was making a lot of progress on their own linker, LLD. What is pretty awesome about LLD is that it's essentially free of any operating system specific code. It's capable of generating binaries for many ELF-based operating systems out of the box, simply by using sane defaults that work well across the board. Compared to GNU Binutils, it also has a more favorable license (MIT vs. GPLv3).

When we started experimenting with LLD, we noticed there were still some blockers that prevented us from using it immediately. An important step during the linking process is that the linker



applies relocations: a series of rules stored in object files that describe how machine code needs to be adjusted to point to the correct addresses of variables and functions when being linked into a program or library. We observed that LLD applied several types of relocations incorrectly, causing resulting executables to access invalid memory addresses almost instantly. This was due to the fact that the LLD developers had mainly focused on getting dynamically linked executables to work, whereas CloudABI uses static linkage.

After filing bug reports and sending various patches upstream, we managed to get LLD working reliably for at least x86-64, i686, and ARM64. For full ARMv6 support we had to wait until LLD 4.0 got released, as ARMv6 uses a custom format for C++ exceptions metadata (EHABI) that LLD didn't yet support.

LLD worked so well for us that at one point we decided to stop using GNU Binutils entirely. Together with Google's Fuchsia operating system, CloudABI is now one of the systems that has switched over to LLD completely. The `devel/cloudabi-toolchain` port now installs a toolchain based on LLVM 4.0, setting up symbolic links for `*-unknown-cloudabi-ld` to point to LLD.

OPERATING SYSTEMS AND EMULATORS

One of the original requirements for running CloudABI programs was that you needed an operating system kernel capable of executing them natively. On FreeBSD, this is very easy to achieve, as FreeBSD 11 and later ship with the kernel modules for that by default (called `cloudabi32.ko` and `cloudabi64.ko`, both depending on common code in `cloudabi.ko`). The Linux kernel patchset has also matured over time, but hasn't been upstreamed, meaning that users still need to install custom-built kernels. On systems like macOS, it's undesirable to install a modified operating system kernel.

To lower the barrier for at least experimenting with CloudABI on these systems, we've developed an emulator capable of running CloudABI executables on top of unmodified UNIX-like operating systems. The emulator works by mapping the executable in the same address space and jumping to its entry point. Code is executed natively, without being interpreted or recompiled dynamically. System calls end up calling into the emulator, which forwards them to the host operating system.

While working on this, we wanted to prevent any complexity in the emulator that could easily be avoided by improving CloudABI itself. For example, CloudABI executables for our 64-bit architectures are now required to be position independent. Whereas systems like HardenedBSD and OpenBSD are mainly interested in using Position Independent Executables (PIE) to allow for Address Space Layout Randomization (ASLR), we see it as a useful tool for guaranteeing that CloudABI executables can be mapped by the emulator without conflicting with address ranges used by the emulator internally.

Another improvement we've made is that CloudABI executables no longer attempt to invoke system calls through special hardware instructions like `int 0x80` and `syscall` directly. This is important, as we don't want CloudABI executables to call into the host system's kernel while emulated. They must call into the emulator instead. The runtime is now required to provide an in-memory shared library (a virtual Dynamic Shared Object, vDSO) to CloudABI executables on startup, exposing one function for every system call supported by the runtime. When running in an emulator, the vDSO points to system call handlers in the emulator. When running natively, the kernel provides a vDSO to the process that contains tiny wrappers that do use the special hardware instructions to force a switch to kernel mode:

```
ENTRY(cloudabi_sys_fd_sync)
  mov $15, %eax
  syscall
  ret
END(cloudabi_sys_fd_sync
```

An advantage of using a vDSO this way is that it makes it a lot easier to add and remove system calls over time. As system calls are now identified by strings, not numbers, third parties can easily place extensions under a custom prefix that doesn't clash with CloudABI's set of system calls (e.g., `acmecorp_sys_*`, as opposed to `cloudabi_sys_*`). Programs can easily detect which system calls are present and absent during startup by simply scanning the vDSO's symbol table.

Finally, we've also made some improvements to the way CloudABI implements Thread-Local Storage (TLS). It is now designed in a way that the emulator can more efficiently switch between the context used by the host and guest process. This is achieved by requiring that the Thread Control Block (TCB) of the guest always retains a pointer to the TLS area of the host. When performing a system call, the emulator can temporarily reinstate its own TLS area by extracting it from the TCB of its guest.

One of the goals behind building an emulator using this approach is that having an easy way of embedding the execution of CloudABI programs is useful for many purposes unrelated to emulation. One can now design a system call tracing utility like `truss(8)` entirely in user space without depending on any special kernel interfaces like `ptrace(2)`. Other interesting use cases include user space deadlock detectors for multithreaded code, and fuzzers to inject random failures.

The user space emulator for CloudABI has in the meantime been integrated into `cloudabi-run` and can easily be enabled by passing in the `-e` command line flag. Below is a transcript of how one can build and run a CloudABI program on macOS.

```
$ cat hello.c
#include <argdata.h>
#include <program.h>
#include <stdio.h>
#include <stdlib.h>

void program_main(const argdata_t *ad) {
  int fd = -1;
  argdata_get_fd(ad, &fd);
  dprintf(fd, "Hello, world!\n");
  exit(0);
}
```

```
$ x86_64-unknown-cloudabi-cc -o hello hello.c
$ cat hello.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
!fd stdout
$ cloudabi-run hello < hello.yaml
Failed to start executable: Exec format error
$ cloudabi-run -e hello < hello.yaml
Hello, world!
```

BETTER C++ SUPPORT

When CloudABI was developed initially, our main focus was to get code written in C to work. After CloudABI's C library became relatively complete and a fair number of packages for software written in C started to appear, we shifted our focus toward improving the experience of porting software written in C++ to CloudABI.

Early on we had managed to get LLVM's C++ runtime libraries (`libcxx`, `libcxxabi`, and `libunwind`) to work, but they still required a lot of local patches. In many cases, these patches were cleanups not specific to CloudABI. They made the code more portable in general. Since the previous article was published, we've been able to get almost all of these patches integrated. At the same time, we've also been able to package Boost, a commonly used framework for C++.

An interesting piece of software written in C++ that we've ported to CloudABI is LevelDB. LevelDB is a library that implements a heavily optimized sorted key-value store, using a data structure called a log-structured merge-tree. It is used within Google as a building block for BigTable, a database system that powers many of their web services.

Porting LevelDB really demonstrated the strength of CloudABI: by omitting any interfaces incompatible with Capsicum, it was trivial for us to find the parts of code that needed to be patched up to work well with sandboxing. In the case of LevelDB, it pointed us straight to `leveldb::Env`, the class that implements all of the filesystem I/O. We've made it possible to use LevelDB in sandboxed software by changing this class to hold a file descriptor of a directory to which filesystem operations should be confined. Whereas you would normally use LevelDB's API to access a database as follows:

```
leveldb::Options opt;
opt.create_if_missing = true;
opt.env = leveldb::Env::Default();
leveldb::DB *db;
leveldb::Status status = leveldb::DB::Open(opt, "/var/...", &db);
db->Put(leveldb::WriteOptions(), "my_key", "my_value");
```

You can make use of LevelDB on CloudABI like this:

```
leveldb::Options opt;
opt.create_if_missing = true;
opt.env = leveldb::Env::DefaultWithDirectory(db_directory_fd);
leveldb::DB *db;
leveldb::Status status = leveldb::DB::Open(opt, ".", &db);
db->Put(leveldb::WriteOptions(), "my_key", "my_value");
```

With libraries like these readily available, we're now able to start working on bringing entire programs to CloudABI. One of the lead developers of Bitcoin, Wladimir van der Laan, happened to discover that most of the libraries used by Bitcoin's reference implementation, like Boost and LevelDB, had already been ported and packaged by us. As a result, Wladimir has been able to successfully port `bitcoind` to CloudABI.

The initial goal of this project is to isolate Bitcoin from other processes running on the same system. A future goal is to use CloudABI to perform privilege separation, so that security flaws in the network protocol handling can't be used by an attacker to obtain direct access to the wallet storing the user's Bitcoins.

RUNNING SANDBOXED PYTHON CODE

In late 2015, I gave a talk about CloudABI at the 32C3 security conference in Hamburg. During this talk, I briefly mentioned that I had plans to port the Python interpreter to CloudABI. This seemed to have made an impression on the audience, as I got an email from Alex Willmer not long after the conference, offering to help out.

During the months that followed, Alex and I worked together a lot, coming up with patches both for Python and CloudABI's C library to get Python to build as cleanly as possible. After getting the interpreter to build, Alex worked on extending Python's module loader, `importlib`, to allow you to include paths to `sys.path` using directory file descriptors, as opposed to using

pathname strings. Finally, I worked on integrating the Python interpreter with Argdata, so that it can be launched through `cloudabi-run`.

Below is a demonstration of what it looks like to run a simple "Hello, world" script using our copy of Python. During its lifetime, the interpreter only has access to Python's modules directory, the script we're trying to execute, and the terminal to which the script should write its message .

```
$ cat hello.py
import io
import sys

stream = io.TextIOWrapper(sys.argdata['terminal'],
    encoding='UTF-8')
print(sys.argdata['message'], file=stream)
$ cat hello.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
path:
- !file
    path: /usr/local/x86_64-unknown-cloudabi/lib/python3.6
script: !file
    path: hello.py
args:
    terminal: !fd stdout
    message: Hello, world!
$ cloudabi-run \
   /usr/local/x86_64-unknown-cloudabi/bin/python3 < hello.yaml
Hello, world!
```

Though it may at first seem somewhat complex to run Python this way, the requirement for injecting all dependencies of Python explicitly does have the advantage that it's now a lot easier to maintain multiple Python environments. Each of these environments may have different versions of third-party modules installed. This approach effectively means there is no longer any need to use Python's own `virtualenv` to achieve isolation between environments. This can already be accomplished by injecting Python with the right set of file descriptors.

The example given above is, of course, only intended to scratch the surface of how you can use CloudABI's copy of Python. On the CloudABI development blog, you can find an article that explains how you can use `socketserver` and `http.server` to build your own sandboxed web services. In the meantime, we're also working on porting the Django web application framework. A preliminary version that is capable of serving requests has already been packaged.

FORMALIZATION OF THE BINARY INTERFACE

All of CloudABI's low-level data types and constants were originally defined through a set of C header files. The problem with these header files was that they became pretty complex over time. As we allow you to run 32-bit CloudABI executables both on 32-bit and 64-bit systems, we had to maintain copies of the definitions that either assumed the target's native pointer size (used by user space) or that assumed 32-bit or 64-bit pointers explicitly (used by kernel space). CloudABI's system call table was translated to FreeBSD's in-kernel format and kept in sync by hand.

To clean this up, Maurice Bos has worked on a project to formalize CloudABI. All CloudABI data types, constants, and system calls are now described in a programming language independent notation in a file called `cloudabi.txt`. Below is an excerpt of what the definitions related to the `cloudabi_sys_fd_read()` system call look like:

```
opaque uint32 fd
  | A file descriptor number.

struct iovec
  | A region of memory for scatter/gather reads.
  range void buf
    | The address and length of the buffer to be filled.

syscall fd_read
  | Reads from a file descriptor.
  in
    fd            fd
      | The file descriptor from which data should be
      | read.
    crange iovec iovs
      | List of scatter/gather vectors where data
      | should be stored.
  out
    size          nread
      | The number of bytes read.
```

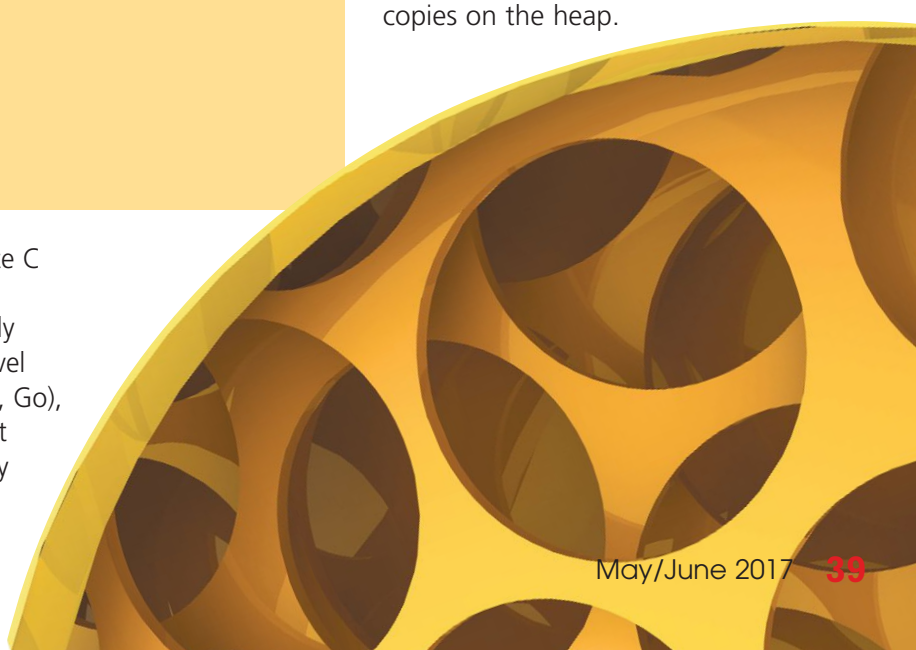
From this file, we now automatically generate C header files, system call tables, vDSOs, and HTML/Markdown documentation. We eventually want to use this framework to generate low-level bindings for other languages as well (e.g., Rust, Go), so that they can be ported to CloudABI without depending on any definitions that are copied by hand.

ARGDATA: NOW A SEPARATE LIBRARY

Though Argdata was originally designed just to be used for passing startup configuration to CloudABI programs, it's actually a pretty flexible and efficient binary serialization library under the hood. Compared to MessagePack, a similar encoding format, it allows for efficient random access of data without performing full deserialization. Compared to `libnv(3)`, a serialization library in FreeBSD's base system, it has a more conventional data model and a simpler API.

Earlier this year, Maurice Bos expressed interest in using Argdata as a format for serializing RPC messages in a non-CloudABI application written in C++. As the Argdata library was tightly integrated into CloudABI's C library, Maurice spent some time making it more portable and turning it into a separate library.

At the same time, he also wrote some really good C++ bindings for Argdata, making use of various features provided by modern revisions of the language (C++11, C++14, C++17). Maps and sequences can be iterated using C++'s range-based for loops. By making use of `std::optional<T>`, C++'s implementation of a '"maybe type," it becomes easier to deal with potential type mismatches. Strings are returned as `std::string_view` objects, meaning they can be used by C++ code without copying them out of the serialized data or allocating copies on the heap.



Below is an example of what the C++ bindings look like when used in practice. When compared to the web server provided in the introduction that uses the C API, the C++ code is a lot more compact and easier to understand.

```
#include <argdata.hpp>
#include <cstdlib>
#include <optional>
#include <program.h>

void program_main(const argdata_t *ad) {
   // Scan through all configuration options and
   // extract values.
   std::optional<int> database_directory, logfile, http_socket;
   for (auto [key, value] : ad->as_map()) {
      if (auto keystr = key->get_str(); keystr) {
         if (*keystr == "database_directory")
           database_directory = value->get_fd();
         else if (*keystr == "logfile")
           logfile = value->get_fd();
         else if (*keystr == "http_socket")
           http_socket = value->get_fd();
      }
   }

   // Terminate if we didn't get started with all
   // necessary descriptors.
   if (!database_directory || !logfile || !http_socket)
      std::exit(1);

   …
}
```

NEXT PROJECT: CLOUDABI FOR KUBERNETES?

Over the last couple years there has been a lot of development in the area of cluster management systems. These systems allow you to treat a large group of servers as a single pool of computing resources on which you can schedule jobs. One system that is gaining popularity is Kubernetes, designed by Google and funded through the Linux Foundation's recently founded Cloud Native Computing Foundation (CNCF). Kubernetes can run any software that has been packaged as a container (using Docker, rkt, etc.).

What I've observed while using Kubernetes is that it has a number of quirks stemming from the fact that it has to be able to run software that is not dependency injected. For example, because programs running in containers are free to bind to arbitrary network ports, every job ("pod") in the cluster must have a unique IPv4 address. This makes Kubernetes consume a lot of network addresses and makes routing tables of nodes in the cluster very complex. Conversely, as most conventional software can only connect to a single network address to reach backend services, Kubernetes does lots of TCP-level load balancing for internal traffic, which makes tracing and debugging very complicated.

The fact that jobs in the cluster are able to create arbitrary network connections also means that security between pods can only be achieved if all containers running in the cluster are configured to make use of cryptography, authentication, and authorization (e.g., using SSL with a per-service custom trust chain). Unfortunately, people hardly ever bother setting that up correctly, meaning it is generally not safe to operate a single Kubernetes cluster for a multitenant environment.

An interesting development for us is that as of version 1.5, Kubernetes no longer communicates with the system's container engine directly. When Kubernetes wants to start a container on a node in the cluster, it sends an RPC for that to an additional process, called the Container Runtime Interface (CRI). This mechanism is intended to allow people to experiment with custom container formats more easily by developing their own CRIs.

In our case, we could implement our own CRI that allows us to run CloudABI processes directly on top of Kubernetes. By using CloudABI, we can enforce jobs running on the cluster to have all of their network connectivity injected by a helper process. This helper process can ensure all traffic in the cluster is encrypted, authorized, and load balanced. Software running on the cluster will no longer need to care about the model of the underlying network.

WRAPPING UP

I hope this article has shown that a lot of interesting things have happened with CloudABI over the last year and that there are even more exciting things planned. As CloudABI is part of FreeBSD 11 and most new features have already been merged into 11-STABLE, CloudABI has become an easy-to-use tool for creating secure and testable software. If you maintain a piece of software that could benefit from this, be sure to experiment with building it for CloudABI.

As most of the discussion around CloudABI takes place on IRC, feel free to join #cloudabi on EFnet if you have any questions or simply want to stay informed about what's going on.

LINKS

CloudABI on FreeBSD: https://nuxi.nl/cloudabi/freebsd/

CloudABI development blog: https://nuxi.nl/blog/

CloudABI on GitHub: https://github.com/NuxiNL

BitCoin for CloudABI: https://laanwj.github.io/

ED SCHOUTEN has been a developer at the FreeBSD Project since 2008. His contributions include FreeBSD 8's SMP-safe TTY layer, the initial import of Clang into FreeBSD 9, and the initial version of the vt(4) console driver that eventually made its way into FreeBSD 10.

CloudABI has been developed by the author's company, Nuxi, based in the Netherlands. It will always be available as open-source software, free of charge. Nuxi offers commercial support, consulting, and training for CloudABI. If you are interested in using CloudABI in any of your products, be sure to get in touch with Nuxi at info@nuxi.nl.