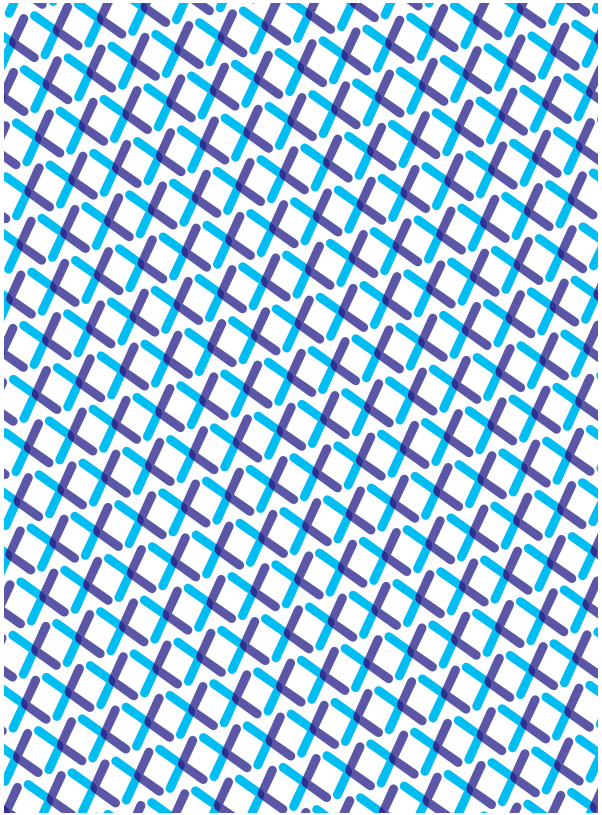

CloudABI: safe, testable and maintainable software for UNIX

Speaker:
Ed Schouten, ed@nuxi.nl

About me

- 2008-present: ed@FreeBSD.org.
  - 2005: Microsoft Xbox 1 port (with rink@).
  - 2008: SMP-safe TTY layer.
  - 2009: First version of `vt(4)`.
  - 2010: ClangBSD.
  - 2011: C11: atomics, Unicode functions.
- 2012-2014: Assimilated by the Borg.
- 2015: Started my own company, Nuxi.
  - Infrastructure for secure and reliable cluster/cloud computing.

Overview

- **What's wrong with UNIX?**
- Short introduction to Capsicum
- CloudABI and Cloudlibc
- Future work

What is wrong with UNIX?

UNIX is awesome, but in my opinion:

- it doesn’t stimulate you to run software securely.
- it doesn’t stimulate you to write reusable and testable software.
- system administration doesn’t scale.

UNIX security problem #1

A web service only needs to interact with:

- incoming network connections for HTTP requests,
- optional: a directory containing data files,
- optional: database backends.

In practice, an attacker can:

- create a tarball of all world-readable data under `/`,
- register `cron` jobs,
- spam TTYs using the `write` tool,
- turn the system into a botnet node.

UNIX security problem #2

Untrusted third-party applications:

- Executing them directly: extremely unsafe.
- Using Jails, Docker, etc.: still quite unsafe.
- Inside a VM: acceptable.

Downside: maintaining Jails and VMs requires more effort.

Reusability and testability

UNIX programs are hard to reuse and test as a whole.

Let’s take a look at how these aspects are solved elsewhere and compare.

Reuse and testing in Java #1

```
class WebServer {
  private Socket socket;
  private String root;
  WebServer() {
    this.socket = new TCPSocket(80);
    this.root = “/var/www”;
  }
}
```

Reuse and testing in Java #2

```
class WebServer {
  private Socket socket;
  private String root;
  WebServer(int port, String root) {
    this.socket = new TCPSocket(port);
    this.root = root;
  }
}
```

Reuse and testing in Java #3

```
class WebServer {
  private Socket socket;
  private Directory root;
  WebServer(Socket socket, Directory root) {
    this.socket = socket;
    this.root = root;
  }
}
```

Reusability and testability

UNIX programs are similar to the first two examples:

- Parameters are hardcoded.
- Parameters are specified in configuration files stored at hardcoded locations.
- Resources are acquired on behalf of you, instead of allowing them to be passed in.

A double standard, compared to how we write code.

Reusable and testable web server

```
#include <sys/socket.h>
#include <unistd.h>

int main() {
     int fd;
     while ((fd = accept(0, NULL, NULL)) >= 0) {
          const char buf[] = “HTTP/1.1 200 OK\r\n”
                             “Content-Type: text/plain\r\n\r\n”
                             “Hello, world\n”;
          write(fd, buf, sizeof(buf) - 1);
          close(fd);
     }
}
```

Reusable and testable web server

Web server is reusable:

- Web server can listen on any address family (IPv4, IPv6), protocol (TCP, SCTP), address and port.
- Spawn more on the same socket for concurrency.

Web server is testable:

- It can be spawned with a UNIX socket. Fake requests can be sent programmatically.

Overview

- What’s wrong with UNIX?
- **Short introduction to Capsicum**
- CloudABI and Cloudlibc
- Future work

Capsicum

Technique available on FreeBSD to sandbox software:

1. Program starts up like a regular UNIX process.
2. Process calls `cap_enter()`.
   - Process can interact with file descriptors. `read()`, `write()`, `accept()`, `openat()`, etc.
   - Process can't interact with global namespaces. `open()`, etc. will return `ENOTCAPABLE`.

Used by `dhclient`, `hastd`, `ping`, `sshd`, `tcpdump`, and various other programs.

My experiences using Capsicum

- **Capsicum is awesome! It works as advertised. Other systems should also support it.**
- ‘Capsicum doesn’t scale’.
  - Porting small shell tools to Capsicum is easy.
  - Porting applications that use external libraries becomes exponentially harder.
- There is no guidance when porting applications.
  - Trial and error until the program works.
- FreeBSD libraries don’t work well with Capsicum.

What does the following code do?

```
/* Timezones. */
localtime_r(&t, &tm);

/* Locales. */
l = newlocale(LC_ALL_MASK, “zh_CN.UTF-8”, 0);
wcstombs_l(buf, L“北京市”, sizeof(buf), l);

/* Random data? */
fd = open(“/dev/urandom”, O_RDONLY);
if (fd == -1) {
     gettimeofday(&tm, NULL);
     pid = getpid();
}
```

Contradicting requirements

- In regular applications we want to load configuration at run time, e.g. from `/usr/share`.
- For Capsicumized binaries it may make more sense to have them compiled in.
- We want functions like `open()`, etc. to be present.
- But throwing a compiler error when used after `cap_enter()` would prevent lots of foot-shooting.

Pure capability-based computing

Thought experiment: having a separate pure capability-based runtime environment.

- Program is always in capabilities mode.
- Capsicum-unsafe functions removed entirely.
  - Causes breakage, but this is good. It's easier to fix the code to build than it is to debug.
- Implementations customized to the environment.
  - Built-in datasets: locales, timezones, `getprotobyname()`, `getservbyname()`, etc.

Implications of pure capabilities

- Safe execution.
  - Less need for virtualization or jails.
  - Simple cloud computing service: run applications for customers instead of offering virtual machines.
- Reusability and testability by default.
  - Just use a different set of file descriptors.
- Dependencies of the application are explicit.
  - Easier release engineering.
  - Higher-level orchestration of software in a cluster.

Overview

- What’s wrong with UNIX?
- Short introduction to Capsicum
- **CloudABI and Cloudlibc**
- Future work

Introducing CloudABI

A specification for a pure capabilities-based runtime:

- POSIX + Capsicum - incompatible features.
- Pretty compact: just 57 system calls.
- Constants, types and structures are defined in a reusable and embeddable format.
  - Support can easily be added to existing operating, similar to `COMPAT_LINUX`.
  - Compile once, run everywhere.
  - Easier to use CloudABI without using the C runtime.

Low-level API

```
/* Allocate memory. */
void *mem;
cloudabi_errno_t error = cloudabi_sys_mem_map(NULL, size,
    CLOUDABI_PROT_READ | CLOUDABI_PROT_WRITE,
    CLOUDABI_MAP_PRIVATE | CLOUDABI_MAP_ANON, -1, &mem);

/* Write to a file. */
cloudabi_ciovec_t iov = { .iov_base = “Hello\n”, .iov_len = 6 };
cloudabi_size_t written;
error = cloudabi_sys_fd_write(fd, &iov, 1, &written);

/* Terminate gracefully. */
cloudabi_sys_proc_exit(123);
```

Low-level API

```
/* Terminate abnormally. */
cloudabi_sys_proc_raise(CLOUDABI_SIGABRT);

/* Obtain random data. */
char buf[100];
cloudabi_errno_t error = cloudabi_sys_random_get(buf, sizeof(buf));

/* Create a directory. */
const char *dirname = “homework”;
error = cloudabi_sys_file_mkdir(fd, dirname, strlen(dirname));
```

Cloudlibc

Cloudlibc is a C library built on top of the low-level API.

- Only contains functions that make sense in a capability-based environment.
  - The goal: 90% POSIX compliant.
  - Compiler errors when using unsupported constructs.
- Very high testing coverage.
  - ~650 unit tests.
  - Good to test the library itself.
  - Also useful to test conformance of the OS.

Contributed code in Cloudlibc

- `malloc()`: jemalloc.
- `<math.h>` and `<complex.h>`: OpenLibm.
  - Portable version of FreeBSD's and OpenBSD's msun.
- Floating point printing and parsing: `double-conversion` library.
  - Uses Florian Loitsch's Grisu algorithm.
  - Supposedly faster than David M. Gay's `gdtoa`.
  - Extensively used by Google (Chrome, V8, Dart).
- IANA tzdata, but not tzcode.

Progress report on Cloudlibc

Complete:

arpa/inet.h assert.h complex.h cpio.h ctype.h dirent.h elf.h errno.h fcntl.h fenv.h float.h iconv.h inttypes.h iso646.h langinfo.h libgen.h limits.h link.h locale.h math.h monetary.h netinet/in.h poll.h pthread.h sched.h semaphore.h setjmp.h signal.h stdalign.h stdarg.h stdatomic.h stdbool.h stddef.h stdint.h stdlib.h stdnoreturn.h strings.h sys/capsicum.h sys/mman.h sys/stat.h sys/time.h sys/types.h sys/uio.h sys/un.h syslog.h tar.h testing.h tgmath.h threads.h time.h uchar.h wctype.h

Mostly done:

stdio.h string.h sys/procdesc.h sys/socket.h unistd.h wchar.h

Progress report on Cloudlibc

In progress:

`aio.h dlfcn.h fnmatch.h netdb.h regex.h sys/event.h`

(Likely) not going to be implemented:

`fmtmsg.h ftw.h glob.h grp.h mqueue.h ndbm.h net/if.h netinet/tcp.h nl_types.h pwd.h search.h spawn.h stropts.h sys/ipc.h sys/msg.h sys/resource.h sys/select.h sys/sem.h sys/shm.h sys/statvfs.h sys/times.h sys/utsname.h sys/wait.h termios.h trace.h ulimit.h utime.h utmpx.h wordexp.h`

Supported platforms

Hardware architectures:

- x86-64

Operating systems:

- FreeBSD: 99.9% of the tests pass.
- NetBSD: 99% of the tests pass.
- Linux: 90% of the tests pass.
- Others: 0% of the tests pass.

How to use CloudABI

1. Install Clang and Binutils, no patches required.
2. Install Cloudlibc.
3. Install additional libraries, such as libc++ for C++14 support.
4. Patch up your operating system kernel to support CloudABI executables.
5. There is no step 5.

Overview

- What’s wrong with UNIX?
- Short introduction to Capsicum
- CloudABI and Cloudlibc
- **Future work**

Future work

- Upstream FreeBSD and NetBSD support.
- Upstream remaining libc++ changes.
- Finish the Linux port.
- Create packages/ports for the Cloudlibc toolchain.
- Have a package manager for standard libraries.
- Design cluster management/orchestration system for running CloudABI processes at a large scale.

More information

CloudABI sources, documentation, etc:

https://github.com/NuxiNL

Contacting Nuxi:

info@nuxi.nl