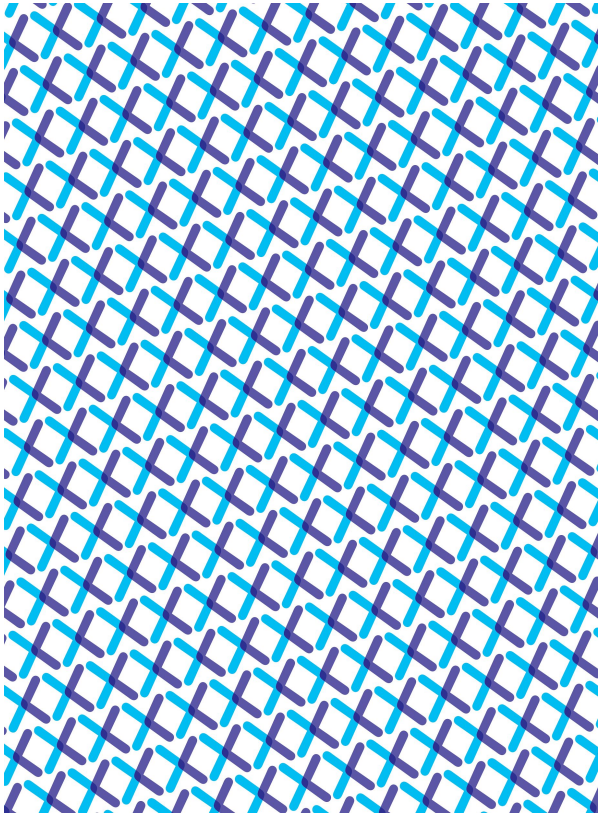

nuxi

CloudABI: safe, testable and maintainable software for UNIX

Speaker:
Ed Schouten, ed@nuxi.nl

Overview

- **What's wrong with UNIX?**
- Introducing CloudABI
- Developing CloudABI software
- Starting CloudABI processes
- Use cases for CloudABI

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.

Access controls: AppArmor

In my opinion not a real solution to the problem:

- Puts the burden of securing applications on package maintainers and end users.
- Application configuration can easily get out of sync with security policy.
- Common solution if security policy doesn’t work: disable AppArmor.

Capabilities: 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.

Experiences using Capsicum

- **Capsicum is awesome! It works as advertised. Other systems should also support it.**
- Code isn’t designed to have system calls disabled.
  - FreeBSD’s C libraries: timezones, locales unusable.
  - Various libraries: non-random PRNG.
  - Heisenbugs, Mandelbugs and Hindenbugs.
- ‘Capsicum doesn’t scale’.
  - Porting small shell tools to Capsicum is easy.
  - Porting applications that use external libraries becomes exponentially harder.

UNIX security problem #2

Untrusted third-party applications:

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

Why can’t UNIX just safely run third-party executables directly? Wasn’t the operating system intended to provide isolation?

Reusability and testability

Claim: UNIX programs are hard to reuse and test.

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 Filesystem root;
  WebServer(Socket socket, Filesystem 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 hard to override global locations.
- Resources are acquired on behalf of you, instead of allowing them to be passed in.

Dependencies are not injected. 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?
- **Introducing CloudABI**
- Developing CloudABI software
- Starting CloudABI processes
- Use cases for CloudABI

Introducing CloudABI

A new UNIX-like runtime environment that allows you to more easily develop:

- software that is better protected against exploits,
- software that is reusable and testable,
- software that can be deployed at large scale.

In a nutshell:
POSIX + Capsicum always enabled - conflicting APIs.

Default rights

By default, CloudABI processes can only perform actions that have no global impact:

- They **can** allocate memory, create pipes, socket pairs, shared memory, etc.
- They **can** spawn threads and subprocesses.
- They **can** interact with clocks (gettimeofday, sleep).
- They **cannot** open paths on disk.
- They **cannot** create network connections.
- They **cannot** observe the global process table.

Additional rights: file descriptors

File descriptors are used to grant additional rights:

- File descriptors to directories: expose parts of the file system to the process.
- Sockets: make a process network accessible.
  - File descriptor passing: receive access to even more resources at run-time.
- Process descriptors: handles to processes.

File descriptors have permission bitmasks, allowing fine-grained limiting of actions performed on them.

Secure web service

Consider a web service running on CloudABI that gets started with the following file descriptors:

- a socket for incoming HTTP requests,
- a read-only file descriptor of a directory, storing the files to be served over the web,
- an append-only file descriptor of a log file.

When exploited, an attacker can do little to no damage.

Cross-platform support

Observation: UNIX ABI becomes tiny if you remove all interfaces that conflict with capability-based security.

- CloudABI only has 58 system calls. Most of them are not that hard to implement.
- Goal: Add support for CloudABI to existing UNIX-like operating systems.
- Allows reuse of binaries without recompilation.
- Mature: FreeBSD and NetBSD, ARM64 and x86-64
- Experimental: Linux

Overview

- What’s wrong with UNIX?
- Introducing CloudABI
- **Developing CloudABI software**
- Starting CloudABI processes
- Use cases for CloudABI

Developing CloudABI software

Building software for CloudABI manually is not easy:

- Cross compiling is hard, not just for CloudABI.
- Toolchain depends on a lot of components.
- Most projects need to be patched in some way:
  - Removal of capability-unaware APIs breaks the build, which is good!
  - cloudlibc tries to cut down on obsolete/unsafe APIs.
  - Autoconf from before 2015-03 doesn’t support CloudABI.

Introducing CloudABI Ports

- Collection of cross compiled libraries and tools.
- Packages are built for FreeBSD, Dragonfly BSD, NetBSD, OpenBSD, Debian and Ubuntu.
  - Native packages, managed through `apt-get`, `pkg`.
  - Contents are identical, except for installation prefix (`/usr` vs. `/usr/local` vs. `/usr/pkg`).
  - Consistent development environment on all systems.
- Packages don't contain any native build tools.
  - Should be provided by the native package collection.
- Packages include Boost, cURL, GLib, LibreSSL, Lua.

CloudABI Ports in action

Install Clang and Binutils from FreeBSD Ports:

```
$ pkg install cloudabi-toolchain
```

Install core libraries from CloudABI Ports:

```
$ vi /etc/pkg/CloudABI.{conf,key}
$ pkg update
$ pkg install x86_64-unknown-cloudabi-cxx-runtime
```

Build a simple application using Clang and cloudlibc:

```
$ x86_64-unknown-cloudabi-cc -o hello hello.c
```

Overview

- What’s wrong with UNIX?
- Introducing CloudABI
- Developing CloudABI software
- **Starting CloudABI processes**
- Use cases for CloudABI

Simple CloudABI program: ls

```
#include <dirent.h>
#include <stdio.h>

int main() {
    DIR *d = fdopendir(0);
    FILE *f = fdopen(1, “w”);
    struct dirent *de;
    while ((de = readdir(d)) != NULL)
        fprintf(f, “%s\n”, de->d_name);
    closedir(d);
    fclose(f);
}
```

Executing our ls through the shell

```
$ x86_64-unknown-cloudabi-cc -o ls ls.c
$ kldload cloudabi64  # FreeBSD ≥ 11.0
$ ./ls < /etc
.
..
fstab
rc.conf
[...]
```

Isn’t there a better way?

Starting processes through the shell feels unnatural:

- The shell cannot (in a portable way) create sockets, shared memory objects, etc.
- How would you know the ordering of the file descriptors that the program expects?
- How do you deal with a variable number of file descriptors?
- You can no longer configure programs through a single configuration file.

Introducing cloudabi-run

```
$ cloudabi-run /my/executable < my-config.yaml
```

- Allows you to start up a CloudABI process with an exact set of file descriptors.
- Merges the concept of program configuration with resource configuration listing.
- Replaces traditional command line arguments by a YAML tree structure.

Configuration for a web server

```
hostname: nuxi.nl
concurrent_connections: 64
listen:
  - 148.251.50.69:80

logfile: /var/log/httpd/nuxi.nl.access.log

rootdir: /var/www/nuxi.nl
```

Configuration for a web server

```
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
hostname: nuxi.nl
concurrent_connections: 64
listen:
  - !socket
    bind: 148.251.50.69:80
logfile: !file
  path: /var/log/httpd/nuxi.nl.access.log
rootdir: !file
  path: /var/www/nuxi.nl
```

Configuration for a web server

```
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
hostname: nuxi.nl
concurrent_connections: 64
listen:
  - !fd 0

logfile: !fd 1

rootdir: !fd 2
```

From a programmer's perspective

```
#include <argdata.h>
#include <program.h>

void program_main(const argdata_t *ad) {
  argdata_get_bool(ad, …);
  argdata_get_fd(ad, …);
  argdata_get_int(ad, …);
  argdata_get_str(ad, …);
  argdata_iterate_map(ad, …);
  argdata_iterate_seq(ad, …);
}
```

Advantages of using cloudabi-run

For users and system administrators:

- Configuring a service requires no additional effort.
- Impossible to invoke programs with the wrong file descriptor layout, as there is no fixed layout.
- No accidental leakage of file descriptors.
- YAML: Easy to generate and process.

For software developers:

- No need to write a configuration file parser.
- No need to write code to acquire resources on startup.

Overview

- What's wrong with UNIX?
- Introducing CloudABI
- Developing CloudABI software
- Starting CloudABI processes
- **Use cases for CloudABI**

Secure hardware appliances

Hardware appliance vendors can run arbitrary code without any compromise to system security:

- Network appliances: users can run custom packet filters without compromising system stability.
- Email appliances: third-party virus scanner and spam filter modules safely.

High-level cluster management

CloudABI as the basis of a cluster management suite:

- Dependencies of software are known up front.
- Allows for smarter scheduling.
  - Automatic capacity planning.
  - Improving locality.
- Automatic migration of processes between systems.
- Automatic routing of traffic on external addresses to internal processes, load balancing, etc.

‘CloudABI as a Service’

A service where customers can upload executables and have them executed in the cloud.

- Unlike Amazon EC2, there is no virtualization overhead.
- Unlike Amazon EC2, there is no need to maintain entire systems; just applications.
- Unlike Google App Engine, applications can be written in any language; not just Python/Java/Go.

More information

CloudABI:

https://github.com/NuxiNL/cloudlibc

https://github.com/NuxiNL/cloudabi-ports

#cloudabi on EFnet

Nuxi, the company behind CloudABI:

https://nuxi.nl/

info@nuxi.nl