Molding Sand: Shaping Permissions of Processes

Kernel Sandboxing & Privilege Separation

About Me

- Emil (any) 🐬
- Studying pure mathematics 🪐
- FLOSS since 2018
  - cURL ⚙️
  - Rosenpass 🐰
  - Tor 🧅
- The White Stripes Connoisseur

Motivation

- I was **a lot** into this in early 2021
- I’ve wrote a privilege separated POP3 daemon
  - Unpublished due to perfectionism at that time
- Apparently some people think that I have expertise with this
- This talk will probably not be perfect
  - I’ve realized that I forgot so much within these three years 👀

Acknowledgements

- Henning Brauer for *OpenNTPD*
- Kristaps Dzonsons for *Bugs Ex Ante*
- Jan Fooken for the title of this talk

Introduction

- Who writes software?
- Who has bugs in their software?
- Who has software that runs on the internet?
- **Your software is broken**
- **People will exploit your broken software in the ugliest ways imaginable**


THIS IS FINE.

What can we do about this?

- Write defensive code
- Get your code audited
- 100% Branch and ~~AC/DC~~ MC/DC coverage
- Use up-to-date libraries
- Formally proof your code
- …
- Ride to work with your unicorn 🦄

This picture has no purpose, it is just a cute tram, choo-choo 🚂

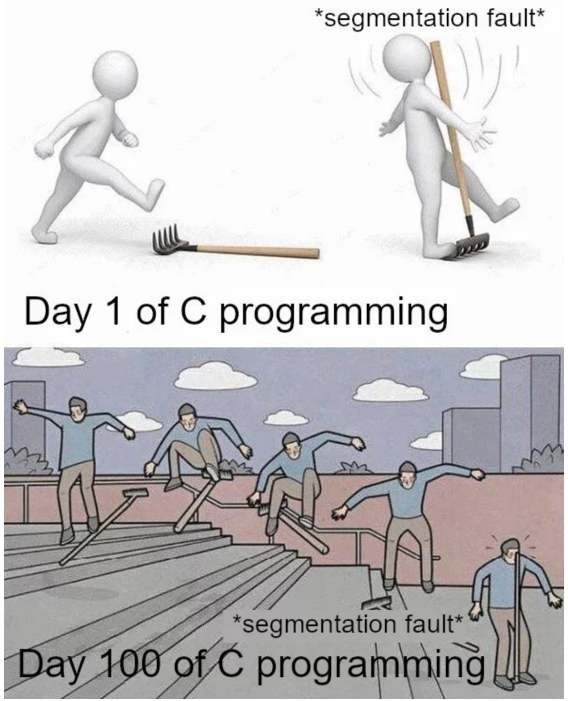
What can we do about this?

- Write defensive code
- ~~Get your code audited~~
- ~~100% Branch and AC/DC MC/DC coverage~~
- ~~Use up-to-date libraries~~
- ~~Formally proof your code~~
- …
- Ride to work with your unicorn 🦄

This picture has no purpose, it is just a cute tram, choo-choo 🚂

Defensive Code

- Use static code analysis tools
  - Such as Rust's borrow checker
- Use fuzzers / Make your code fuzzable
- **Use APIs provided by the operating system**
  - It WILL NOT prevent vulnerabilities
  - It CAN prevent the possible damage
- **Structure your program logic in separate units**
- This talk will not cover applied sandboxing mechanisms, such as `systemd-analyze security`



Demotivative Example

File-System Permissions

- Introduced in 1961 in CTSS
- Everyone™ has screwed up this at least once
- Octal codes are hard to understand
  - Mostly muscle memory
- **We still fail after more than 60 years!**
- **Security can be hard**

The Berlin Wall was built the same year

Kernel Sandboxing APIs

Use of Kernel Sandboxing APIs

- Certain APIs are very complicated and pretty unportable
- Operate on processes and their future descendants
- Easy in C and Zig
- Doable in Rust and C++
- Almost impossible in Go, Java, and Python
- **The closer to the OS, the easier it gets**
- **What can an attacker do, when they gain full control over the process?**

Two Types of Sandboxing APIs*

Limitation

- Removes capabilities from a process
- Restricts system calls
  - In their general availability
  - In their arguments
  - In their behavior

Isolation

- Removes visibility from process
- Hides OS resources from the process
  - Files, Folders, Subtress of the FS
  - Network interfaces
  - Other processes
  - …

*: These terms were coined by me, they are not used by anyone else

Overview of sandboxing APIs

1971	setuid(2)
1979	chroot(2)
1999	capabilities(7)
2000	jail(2)
2005	seccomp(2)
2012	capsicum(4)
2015	pledge(2)
2018	unveil(2)
2021	landlock(7)

`setuid(2)` – Isolation

- Exists since the first UNIX version
- `setuid(2)` ≠ setuid bit
- Changes the owner of a process
    - Requires root
- Foundation for *privilege revocation*
    - A root process changing the process owner to a normal user, thereby dropping all of its privileges
- Still common today, although advanced by real and effective UID
- **Use Case**: Program only needs root during initialization

John Lennon's *Imagine* was released in 1971

```c
int
main(void)
{
    const uid_t NGINX_USR= 42;
    const gid_t NGINX_GRP= 42;

    /* Create, bind, and listen on socket(2) */

    if (setuid(NGINX_USR) == -1 || seteuid(NGINX_USR) == -1) {
        err(1, "setuid");
    }
    if (setgid(NGINX_GRP) == -1 || setegid(NGINX_GRP) == -1) {
        err(1, "setgid");
    }

    /* Handle requests, ... */

    return 0;
}
```

Sample output of `setuid(2)`

```
engler@thecure ~ % ps aux | grep a\.out
engler            1736   0.0  0.0 410733312   1472 s000  S+    4:20PM   0:00.00 grep a.out
root              1734   0.0  0.0 410592944   1120 s002  S+    4:20PM   0:00.00 ./a.out
root              1733   0.0  0.0 410791024  12736 s002  S+    4:20PM   0:00.03 sudo
./a.out
engler@thecure ~ % ps aux | grep a\.out
engler            1738   0.0  0.0 410733312   1472 s000  S+    4:20PM   0:00.00 grep a.out
engler            1734   0.0  0.0 410601136   1152 s002  S+    4:20PM   0:00.00 ./a.out
root              1733   0.0  0.0 410790464  12688 s002  S+    4:20PM   0:00.03 sudo
./a.out
```

chroot(2) - Isolation

- **Convenience mechanism, sometimes abused as a sandbox**
- Changes the root directory of a process and all its future children
- Does not affect already opened file descriptors
- Hard to use securely, wrong usage opens new vulnerabilities
    - That’s why it requires root
- **Use Case:** Uhmmm? Convenience? 👉👈

```c
int
main(void)
{
    chroot("sandbox/");
    chdir("../../../");
    chroot(".");
}
```

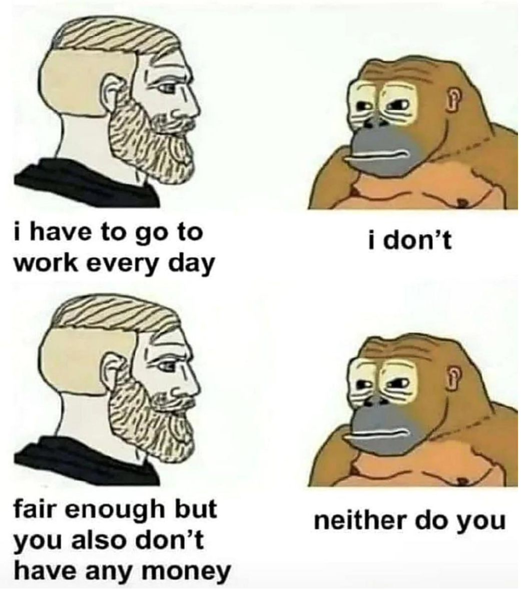
`chroot(2)` **does not change the current working directory!**


THIS IS FINE.

```c
int
main(void)
{
    chroot("sandbox/");
    chdir("/");

    chdir("../../../");
    chroot(".");
}
```

Using `chroot(2)` securely

- **Don't use `chroot(2)`**
- **Use Case**: Process never needs file system access
- The chroot directory must be empty and owned by root
- Many systems provide `/var/empty` for this



Unrelated shitpost

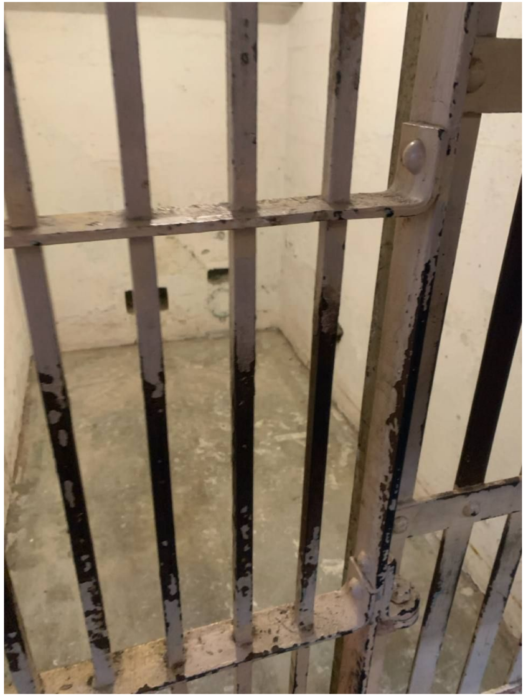
`capabilities(7)` – Limitation

- Introduced around 2000 in Linux
- Can be used externally and by processes themselves
- Associates each root operation with a certain capability
  - `CAP_SYS_ADMIN` – Making all of this pointless
  - `CAP_NET_RAW` – Creating raw sockets
  - `CAP_SYS_CHROOT` – Using `chroot(2)`
  - ...
- Process runs as root but behaves like a normal user in operations uncovered by its capabilities
- **Use Case**: Process only needs a subset of root privileges until termination
  - Example: NTP Client

```c
int
main(void)
{
	cap_t		caps;
	cap_value_t	required_caps[1] = { CAP_SYS_CHROOT };

	/* Allocate our capability list. */
	caps = cap_init();
	assert(caps != NULL);

	/* Add the capabilities we need to caps. */
	assert(cap_set_flag(caps, CAP_PERMITTED,  1, required_caps, CAP_SET) != -1);
	assert(cap_set_flag(caps, CAP_EFFECTIVE,  1, required_caps, CAP_SET) != -1);

	/* Apply it. */
	assert(cap_set_proc(caps) != -1);

	/* Free no longer required resources. */
	cap_free(caps);

	/* chroot(2) will work. */
	assert(chroot("/") != -1);

	/* Rebooting the system will not. */
	assert(syscall(SYS_reboot, 0xfee1dead, 0x28121969, LINUX_REBOOT_CMD_POWER_OFF) == -1);
	assert(syscall(SYS_reboot, 0xfee1dead, 0x05121996, LINUX_REBOOT_CMD_POWER_OFF) == -1);
	assert(syscall(SYS_reboot, 0xfee1dead, 0x16041998, LINUX_REBOOT_CMD_POWER_OFF) == -1);
	assert(syscall(SYS_reboot, 0xfee1dead, 0x20112000, LINUX_REBOOT_CMD_POWER_OFF) == -1);

	return 0;
```

`jail(2)` – Isolation

- Introduced in 2000 by Poul-Henning Kamp
- Essentially `chroot(2)` but for real isolation
  - OG-Container Solution, 10 years older than Docker
  - Can isolate an entire subtree from the system
- Can be used externally and by processes themselves
- Usually requires root
- **Use Case**: Isolate like a VM but without the overhead



Prison or smth. idk about jails

Isn’t this like Linux `namespaces(7)`?

- Both offer similar end-goals and were introduced around the same time
- `namespaces(7)` is more fragmented, instead of monolithic
  - PID namespace, NET namespace, MNT namespace, …
- **Jails start from full isolation that can be reduced**
- **`namespaces(7)` start from zero isolation that can be built up**
- `namespaces(7)` is harder to use
  - FS isolation requires around seven steps to perform
  - Network namespace is still barely documented
  - …

```c
int
main(void)
{
        struct jail     jail_cfg = {
                .version = JAIL_API_VERSION,
                .path = "/root/of/jail",
                .hostname = "example",
                .jailname = "example jail",
                .ip4s = 0,
                .ip6s = 0,
                .ip4 = NULL,
                .ip6 = NULL
        };

        if (jail(&jail_cfg) == -1) {
                err(1, "jail");
        }

        puts("I AM JAILED :3");

        return 0;
}
```

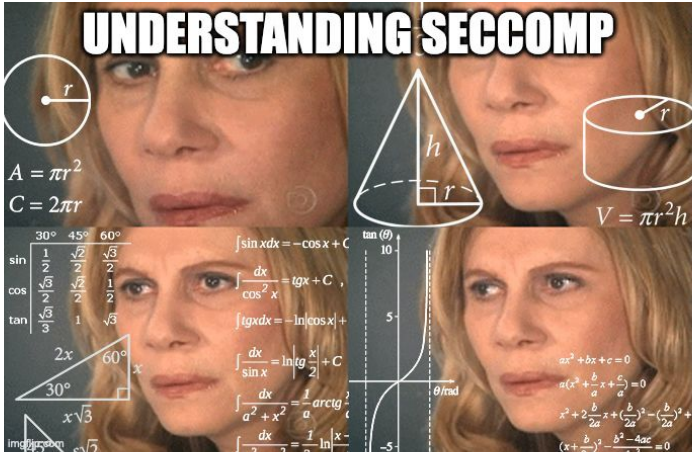
Warning: Deprecated

`seccomp(2)` – Limitation

- Introduced in 2005 in Linux
- Provides a whitelist feature for system calls
  - Whitelist filters entire system calls as well as arguments
  - Whitelist may never be expanded
- Violation will result in `SIGKILL`
- Each system call has to be whitelisted manually
  - Provides very high security at the cost of very high complexity
- Unportable
  - Interfacing application have to take the libc and the architecture into account
  - Example: `fork(2)` on glibc
- **Use Case**: Process only needs a specific set of system calls

`seccomp(2)` – No example unfortunately

- Due to its complexity, a minimal example would not fit onto a slide
- Have a look at the `seccomp(2)` manual page instead, it contains a minimal example :)



capsicum(4) – Limitation and Isolation

- Introduced in 2012 for FreeBSD
- Capabilities by FDs rather than processes
- Processes are placed into capability mode
  - Only system calls with file descriptors are allowed (from this point onward)
  - Each file descriptor has different capabilities
- File descriptors are created with full privilege which might be reduced
- Example: File descriptor may be `read(2)` and `write(2)` but not `fchmod(2)`
- **Use Case**: Isolate resources and limit capabilities

capsicum(4) - Limitation and Isolation

```c
int
main(void)
{
	cap_rights_t rights;
	char         buf[64];
	int          dir_fd, fd;

	/* Open a directory before we enter the sandbox. */
	dir_fd = open("/home/engler/sandbox" , O_RDONLY |
O_DIRECTORY);
	assert(dir_fd != -1);

	/* Enter the sandbox. */
	assert(cap_enter() != -1);

	/* We can no longer create file descriptors. */
	assert(open("/", O_RDONLY) == -1);

	/* Open file for RW in the sandbox. */
	fd = openat(dir_fd, "foo", O_RDWR);
	assert(fd != -1);

	/* Limit the permissions. */
	cap_rights_init( &rights, CAP_READ);
	assert(cap_rights_limit(fd, &rights) != -1);

	/* Read will work. */
	assert(read(fd, buf, sizeof(buf)) > 0);
	/* Write will not. */
	assert(write(fd, "Meow :3", 7) == -1);
	/* Seek will not. */
	assert(lseek(fd, 1, SEEK_SET) == -1);

	return 0;
}
```

1. Open the directories that shall be available
2. Enter the sandbox
3. Open files in the sandbox
4. Restrict these files

`pledge(2)` - Limitation

- Introduced by OpenBSD in 2015
- **Very easy to use, yet very secure!**
- **A single function with two parameters!**
- System calls are grouped into categories
  - stdio, rpath, wpath, inet, …
- Process whitelists these system call categories
  - Categories also influence behavior of certain system calls
  - Once a privilege has been taken away, it can never be gained back
- Using a forbidden system call results in `SIGKILL`
- **Use Case**: Process only needs a specific set of system calls

SerenityOS also supports `pledge(2)`

```c
int
main(void)
{
	if (pledge("stdio rpath inet", "") == -1)
		err(1, "pledge");

	/* Read configuration file ... */

	if (pledge("stdio inet", "") == -1)
		err(1, "pledge");

	/* Do webserver stuff. */

	return 0;
}
```

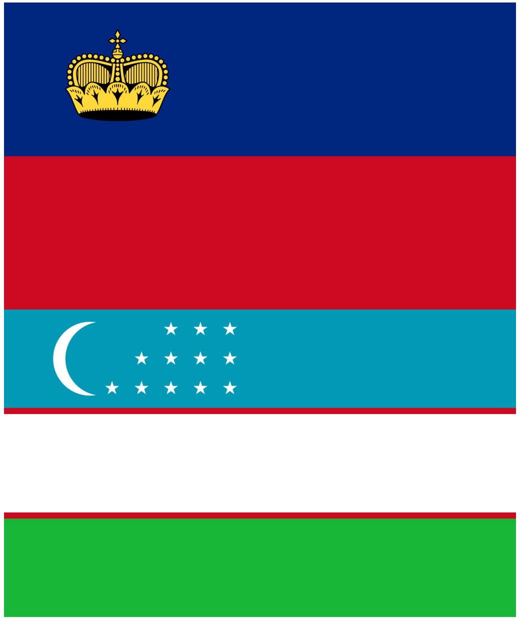
unveil(2) - Isolation

- Introduced in OpenBSD in 2018
- Removes visibility of the entire filesystem
- Process calls unveil(2) to make certain paths with certain permissions visible
- Once a set of path has been established, this function will be disabled
- Already achieves a great level of security
- **Use Case**: Application only needs certain paths in the file system

```c
int
main(void)
{
    /* Make two files visible. */
    if (unveil("/home/engler/config", "r") == -1)
        err(1, "unveil");
    if (unveil("/home/engler/log", "w") == -1)
        err(1, "unveil");

    /* Prevent future calls to unveil(2). */
    if (unveil(NULL, NULL) == -1)
        err(1, "unveil");

    /* Paths not unveiled cannot be opened (file not found). */
    assert(open("/root/.ssh/authorized_keys", O_APPEND) == -1);

    /* Future calls to unveil(2) will now fail. */
    assert(unveil("/usr/bin/sudo", "rx") == -1);
}
```

landlock(7) – Isolation

- De facto `unveil(2)` for Linux, introduced in 2021
  - Still no libc wrapper 😔
- Application specifies a global ruleset of permissions
- Application gives each path a subset of permissions
- More fine grained control with directories
  - Permissions to only create files, symlinks, sockets, …
- **Use Case**: Application only needs certain paths in the file system



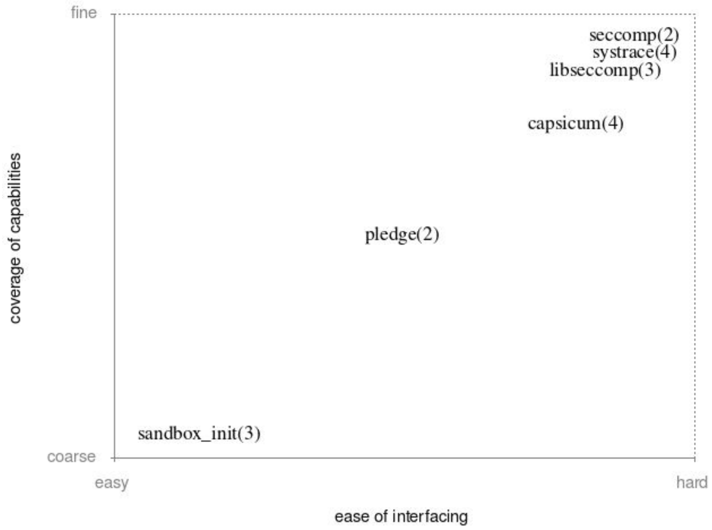
Liechtenstein and Uzbekistan are the only doubly landlocked countries in the world

```c
int
main(void)
{
        struct landlock_ruleset_attr                  attr = {0};
        struct landlock_path_beneath_attr    rule;
        int                          uleset_fd, fd_cfg, fd_log;

        /* Set of available privileges for a file. */
        attr.handled_access_fs  =
                LANDLOCK_ACCESS_FS_READ_FILE  |
                LANDLOCK_ACCESS_FS_WRITE_FILE;
        ruleset_fd  = landlock_create_ruleset( &attr, sizeof(attr), 0);
        assert(ruleset_fd  != -1);

        /* Open the files as paths. */
        fd_cfg  = open("/home/engler/config" , O_PATH);
        fd_log  = open("/home/engler/log" , O_PATH);
        assert(fd_cfg  != -1 && fd_log  != -1);

        /* Configure permissions. */
        rule.allowed_access  = LANDLOCK_ACCESS_FS_READ_FILE;
        rule.parent_fd  = fd_cfg;
        assert(landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,    &rule, 0) !=
-1);
        close(fd_cfg);

        rule.allowed_access  = LANDLOCK_ACCESS_FS_WRITE_FILE;
        rule.parent_fd  = fd_log;
        assert(landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,    &rule, 0) !=
-1);
        close(fd_log);

        /* Prevent more privileges. */
        assert(prctl(PR_SET_NO_NEW_PRIVS,  1, 0, 0, 0) != -1);

        /* Apply permissions. */
        assert(landlock_restrict_self(ruleset_fd,   0) != -1);

        /* Open (and close) the files. */
        fd_cfg  = open("/home/engler/config" , O_RDONLY);
        assert(fd_cfg  != -1);
        close(fd_cfg);

        fd_log  = open("/home/engler/log" , O_WRONLY);
        assert(fd_log  != -1);
        close(fd_log);

        assert(open( "/home/engler/config" , O_WRONLY)  == -1);
        assert(open( "/home/engler/log" , O_RDONLY)  == -1);
        assert(open( "/root/.ssh/authorized_keys" , O_APPEND)  == -1);
```

1. Define set of available permissions
2. Configure permissions for each path
3. Prevent new paths to be allowed
4. Enter the sandbox

Summary

- “Complexity is the worst enemy of security.” – Bruce Schneier
- Most technologies are terribly complex and over-engineered
    - `pledge(2)` and `unveil(2)` being the exception
- Why is it so bad?
    - ~~The NSA tries to keep systems insecure~~
    - ~~Big companies do gatekeeping to sell support~~



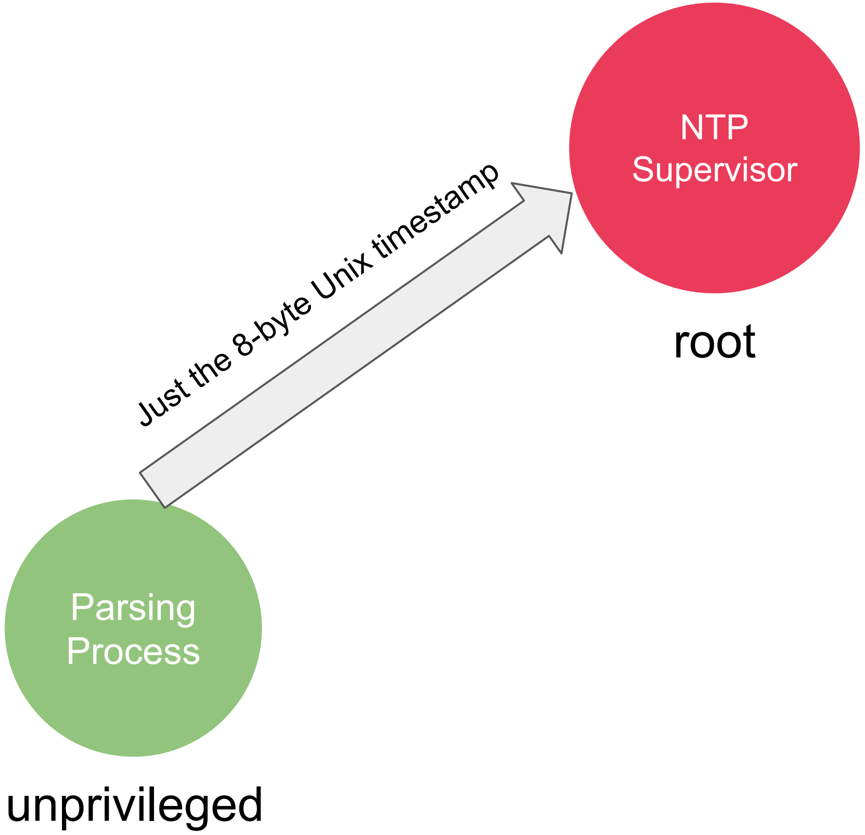
Overview of security mitigations
© Kristaps Dzonsons

Privilege Separation

Privilege Separation – The Motivation

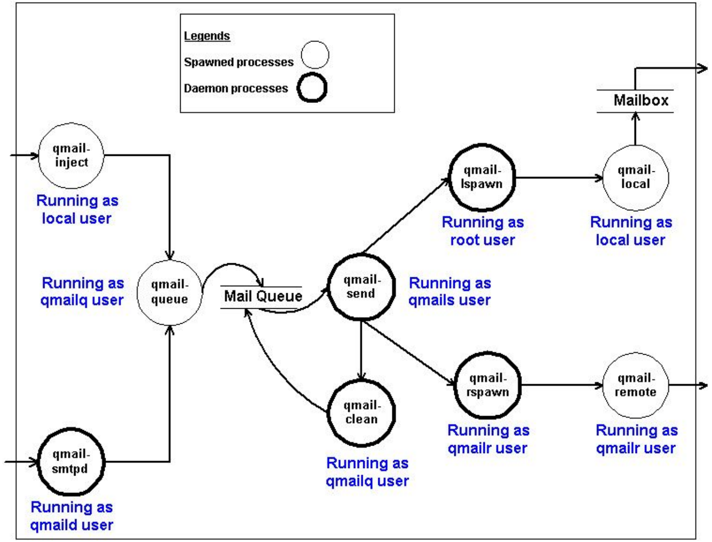
- Sandboxing is usually on process level
- A big monolithic process with lots of privileges is not helpful
- Idea: Use child processes and let each of them just do a single task
  - Aligns very well with the Unix philosophy
  - Much more fine grained privilege control


NTP
Daemon
root


NTP Supervisor
root
Just the 8-byte Unix timestamp
Parsing Process
unprivileged

History of Privilege Separation

- Preliminary work by djb in 1995 with qmail
  - Several small processes composing a complete SMTP server
  - One of them as root, two of them as local user, the rest fully unprivileged
- Initial implementation in 2002 for OpenSSH
  - Unprivileged child process that process all network data
  - Communication happens is achieved by pipes
  - Authentication only happens when child AND parent agree → corrupted child will not lead to access



The qmail Process Architecture
© Ralph Johnson

`fork(2)` – Creating children in Unix

- Processes are structured as trees
- Process can `fork(2)` to create exact copy of itself
  - Copies file descriptors, heap allocations, variables, …
- `fork(2)` branches control flow into parent and child
  - Parent’s result is the PID of the child
  - Child’s result is `0`
- Orphaned processes get PID1 as new parent
- Leads to funny Google searches such as “How to remove childeren from parent using `fork(2)`?”

```c
int
main(void)
{
	pid_t child;

	switch ((child = fork())) {
	case -1:
		/* error */
		err(1, "fork");
	case 0:
		/* child */
		_exit(0);
	default:
		/* parent */
		_exit(0);
	}

	return 0;
}
```

Inter-Process Communication

- Unix offers gazillion ways for IPC
  - Signals, Sockets, Pipes, Shared Memory, Filesystem, …
- Unix Domain Sockets are usually the best choice
  - Very fast, usually up to 500MB/s
  - Allow file descriptor passing (other end receives copy of file descriptor)
  - Atomic in nature
  - Messages can be distinct datagrams easily distinguishable

```
int
main(void)
{
	int   sockets[2], parent, child;
	char  buf[64];
	pid_t child;

	if (socketpair(AF_UNIX, SOCK_DGRAM,0, sockets) == -1) {
		err(1, "socketpair");
	}
	parent = socket[0];
	child = socket[1];

	switch (fork()) {
	case -1:
		/* error */
		err(1, "fork");
	case 0:
		/* child */
		close(child);
		send(parent, "meow :3", 7, 0);
		_exit(0);
	default:
		/* parent */
		close(parent);
		recv(child, buf, 64, 0);
		assert(memcmp(buf, "meow :3", 7) == 0);
		_exit(0);
	}

	return 0;
}
```

Inter-Process Communication – Use a library!

- **You do not want to use sockets without a library!**
- Libraries usually provide:
  - A generic message format with header and payload
  - Guarantees that messages are received in order and as a whole
  - Buffering around I/O
  - Abstraction around file descriptor passing (doing this by hand is as terrible as using `ptrace(2)`)
- Implementing all of this by hand is a nuisance
- Possible libraries: imsg, zeromq, …
- I like imsg from OpenBSD, because it is portable and only ~500 LOC

Case Study: OpenNTPD

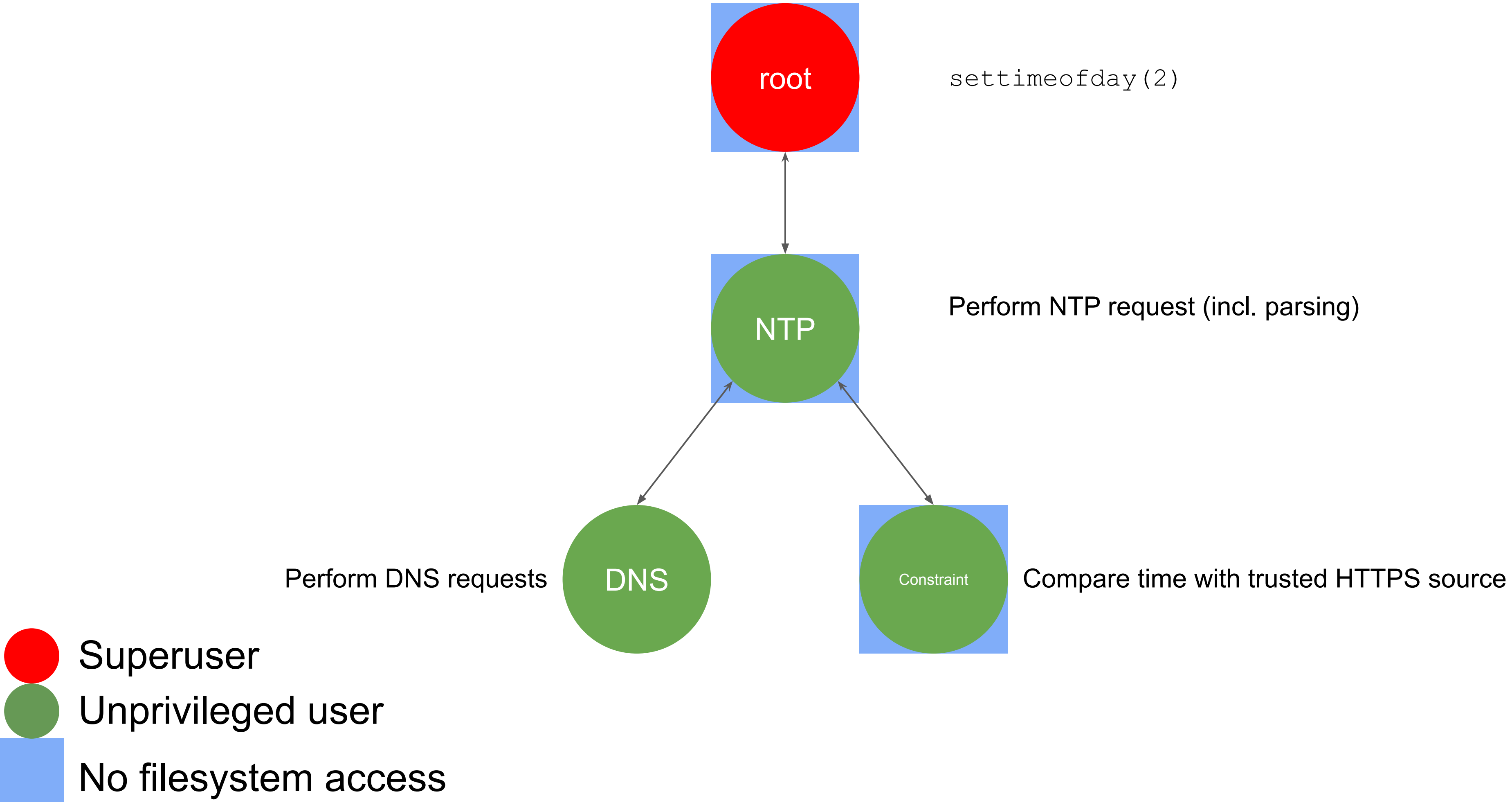
Why study an NTP daemon?

- **NTP Daemons offer the perfect attack surface!**
- Implement an insecure protocol from 1985
- Need root privileges all the time
- Usually start as one of the earliest processes
- Usually run 24/7
- A remote code execution here could have disastrous consequences

Hall & Oates released *Out Of Touch* in 1985

Why study OpenNTPD

- Implements privilege separation very well
- Only about 4000 LOC
- Only one CVE in 21 years
- Very clean code
- I had a friendly e-mail thread with its author 😁
  - Moin Henning! 👋


root
settimeofday(2)
NTP
Perform NTP request (incl. parsing)
Perform DNS requests
DNS
Constraint
Compare time with trusted HTTPS source
Superuser
Unprivileged user
No filesystem access

Thank You!