Sandbox Your Program Using FreeBSD's Capsicum

Jake Freeland
jfree@FreeBSD.org

BSDCan 2025

About Me

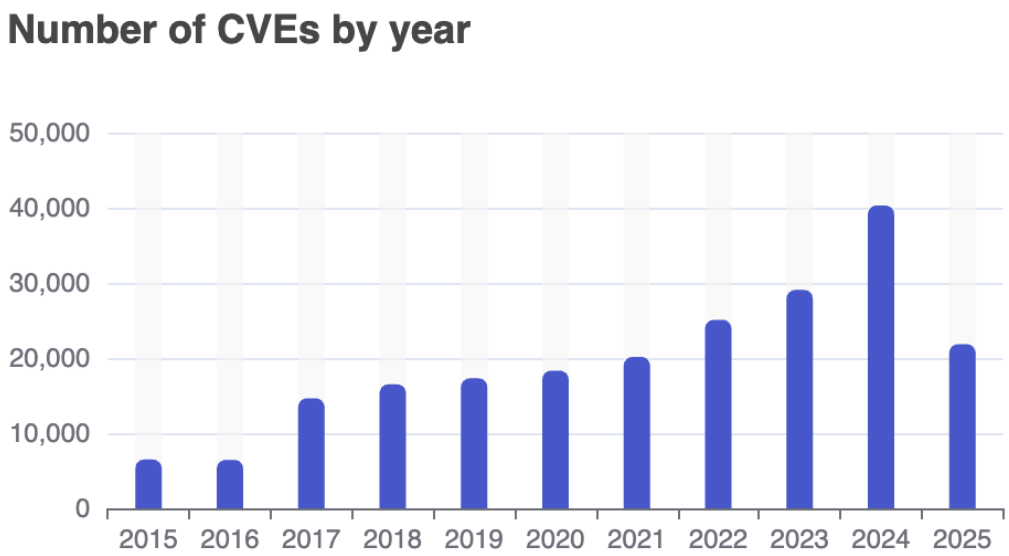
Jake Freeland - jfree@FreeBSD.org

FreeBSD source committer, focusing on porting software from Linux to FreeBSD and vice versa.

I work at NIKSUN where I maintain a driver stack for custom networking hardware that is capable of packet capture and live analysis at 100Gbps speeds.

Why Care About Program Security

CVEs are on the rise:



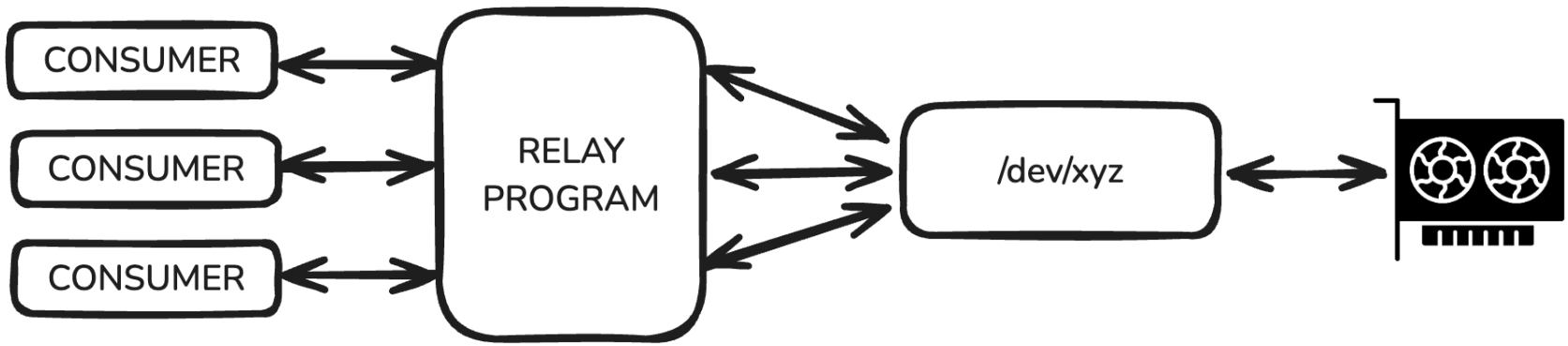
source: `cvedetails.com/browse-by-date.php`

IBM's Cost of a Data Breach Report 2024 shows that global average cost of a breach is 4.88 million USD, up 10% over the previous year.

Example Program

Let's start with an example program that we would like to harden.

Consider a program that relays messages between a set of unprivileged consumer processes and a kernel device.



Program Security

Based on our program's responsibilities, it is probably going to have to be a daemon.

A daemon running in the background is more suceptible to attacks since it doesn't have a limited runtime window. An attack can be issued at any time.

This introduces some important security concerns, especially if the process is run under the root user.

Let's look at security models that could protect our program.

Security Models: None

The easiest security model is to run our daemon under the root user, giving it access to all necessary resources. This is a suboptimal model because we'd be giving the daemon access to more resources than it really needs.

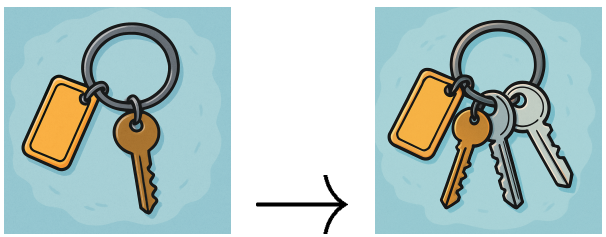
If an exploit that leads to code execution is found, then the attackers could potentially obtain unrestricted access to the underlying system.

Security Models: POSIX 1003.1e Capabilities

Another approach would involve running the daemon as an unprivileged user, but with POSIX 1003.1e capabilities.

In short, the POSIX 1003.1e standard splits root privileges up into more granular pieces called capabilities. Individual capabilities can be assigned to specific processes to give them root-like privileges in a specific kernel subsystem.

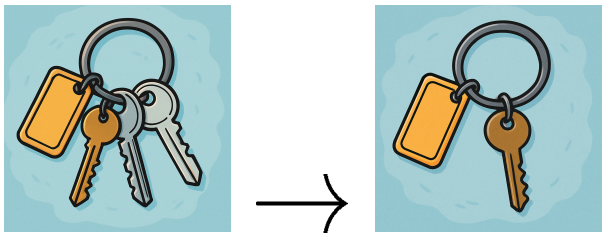
For example, processes with the `CAP_KILL` capability can bypass permission checks for sending signals.



Security Models: Pledge

A similar kind of privilege separation is provided in OpenBSD's `pledge(2)` framework. Unlike POSIX 1003.1e capabilities, pledge narrows the calling process' privileges based on a provided set of *promises* instead of expanding them.

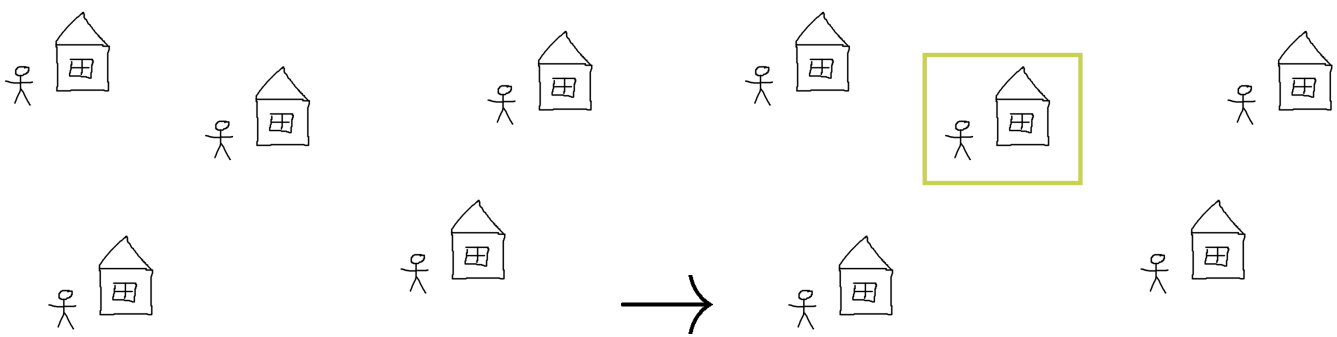
With `pledge(2)`, our relay program could run as the root user, but with restricted access to all kernel subsystems that weren't specified in the *promises* list.



Security Models: Capsicum

Capsicum provides a security sandbox called capability mode. Once a program enters capability mode, access to new resources and interprocess communication are severely curtailed.

The restrictive nature of sandboxes can seem daunting, but they offer the most granular protection out of the other models mentioned.

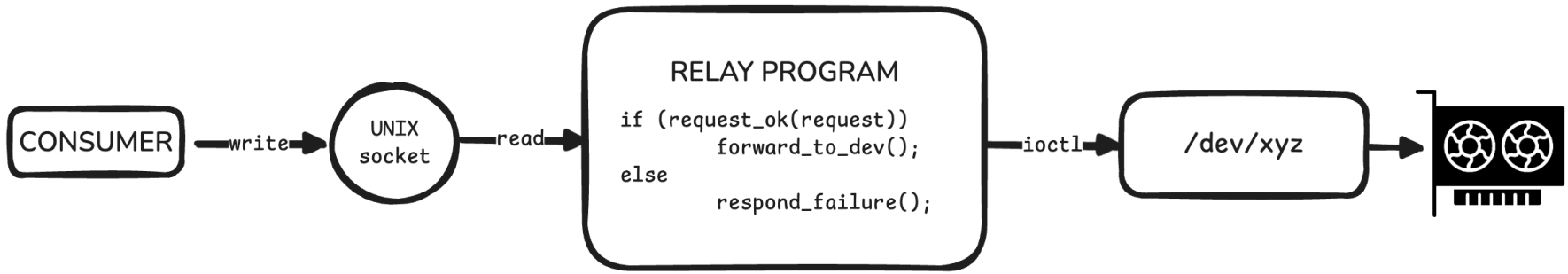


Example Program, Continued

Now that we know our options, let's analyze our example problem more and identify what models work best.

Example Program, Continued

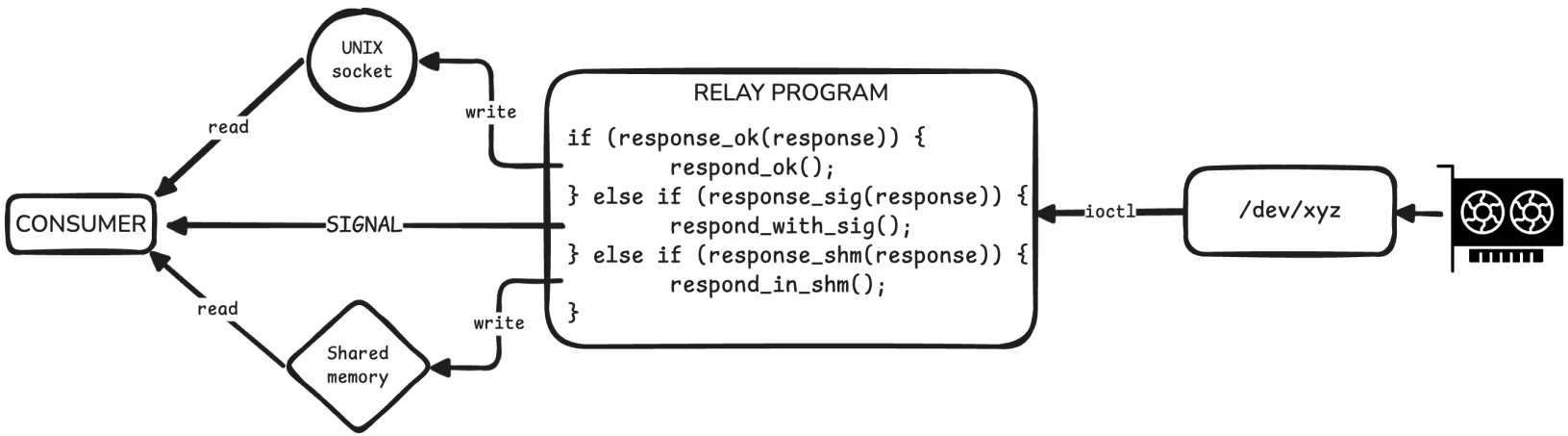
Unprivileged consumer processes send messages over a UNIX socket to the relay program. If the consumer's message is approved by the relay, a relevant `ioctl(2)` call will be made to `/dev/xyz`.



Example Program, Continued

Once the target device processes a message, it will send a response back to our program. Depending on the response, our program will take different actions. It may:

- Relay the response back to the requesting process over the UNIX socket
- Send a signal to the requesting process
- Write data to a temporary shared memory region, whose path is provided by the requesting process



Example Program Resource Requirements

Based on the responsibilities of our program, let's list out required privileges:

- Read and write access to the target device's `devfs(5)` entry to send and receieve messages.
- Read and write access to the UNIX socket to read relay requests and respond.
- Ability to send signals to a requesting process
- Write access to arbitrary shared memory regions

Security Models: Comparison

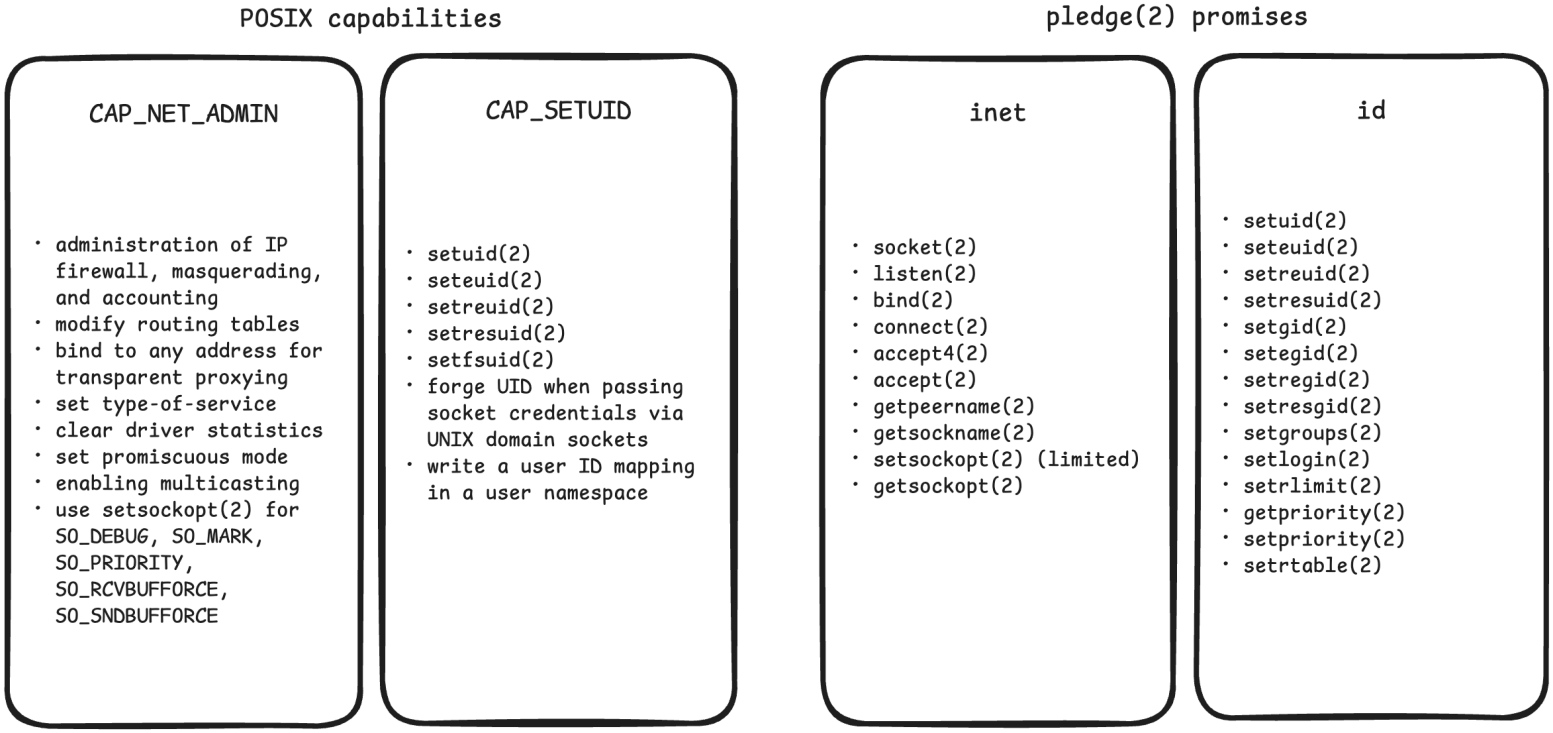
Support Per Security Model				
	As root	POSIX capabilities	Pledge promises	Capsicum
Device R/W	Yes	`CAP_FOWNER`	`unix`	Device pre-open
Socket R/W	Yes	`CAP_FOWNER`	`unix`	Socket pre-open
Signals	Yes	`CAP_KILL`	`proc`	Casper sub-process
Shared memory	Yes	`CAP_IPC_OWNER`	No	Casper sub-process

Privilege Silos

POSIX capabilities and pledge promsies are functionally very different, but both mechanisms attempt to break up privileges into subgroups.

Each subgroup is a collection of related calls that are grouped together under a single security classification. You can think of these groups as privilege silos.

Privilege Silos



If your program needs privileges from a silo, then it is forced to take all other privileges in that silo, even if it doesn't need them.

Capsicum Approach

There are no privilege silos like this in Capsicum. Instead, resource acquisition is done in other ways.

Let's get into Capsicumization, or the process of reworking a program to execute in a Capsicum capability sandbox.

Capsicumization

On systems that support Capsicum, a program may be sandboxed using `cap_enter(2)`:

```
#include <sys/capsicum.h>
#include <stdio.h>

int
main(void)
{
    /* Enter Capsicum's capability mode. */
    cap_enter();
    printf("Hello world from capability mode\n");
}
```

Once a program enters capability mode, it will not be able to acquire new resources by itself.

For example, opening a file using `open(2)` will trigger a capability violation, causing the call to to fail and set errno to `ECAPMODE`: `Not permitted in capability mode.`

Capsicum Resource Acquisition

Processes in capability mode can continue to use resources that were acquired before `cap_enter(2)`.

For this reason, the easiest form of Capsicumization is to open required resources before entering the sandbox:

```
foofd = open("/home/jfree/foo", O_RDONLY);

cap_enter();

if (read(foofd, buf, sizeof(buf)) != -1)
        printf("Success!\n");
```

Capsicum Resource Acquisition

Sometimes you don't know what you need before you enter the sandbox.

If a program requires access to an unknown number of resources that exist in a certain subdomain, then the openat(2), mkdirat(2), bindat(2), and other *at() system calls may be useful:

```
dirfd = open("/home/jfree", O_RDONLY | O_DIRECTORY);

cap_enter();

/* Open "/home/jfree/bar". */
if (openat(dirfd, "bar", O_RDONLY) != -1)
        printf("Success\n");
```

Capsicum Resource Acquisition Restrictions

It is not possible to access resources outside of the subdirectory domain provided by a relative reference though.

```
dirfd = open("/home/jfree", O_RDONLY | O_DIRECTORY);
cap_enter();

/* Open "/home/beastie". */
if (openat(dirfd, "../beastie", O_RDONLY) < 0)
        printf("Failure\n");
```

Capsicum Pre-Opening

In short, pre-opening resources before doing `cap_enter(2)` will allow your program to continue using them normally.

For simple programs, this is an effective approach to begin Capsicumization.

Capsicum Pre-Opening Caveat

Pre-opening resources works great for programs that have predictable resource requirements, but some programs require resources on-demand.

In this case, the developer can opt to only sandbox specific parts of their program.

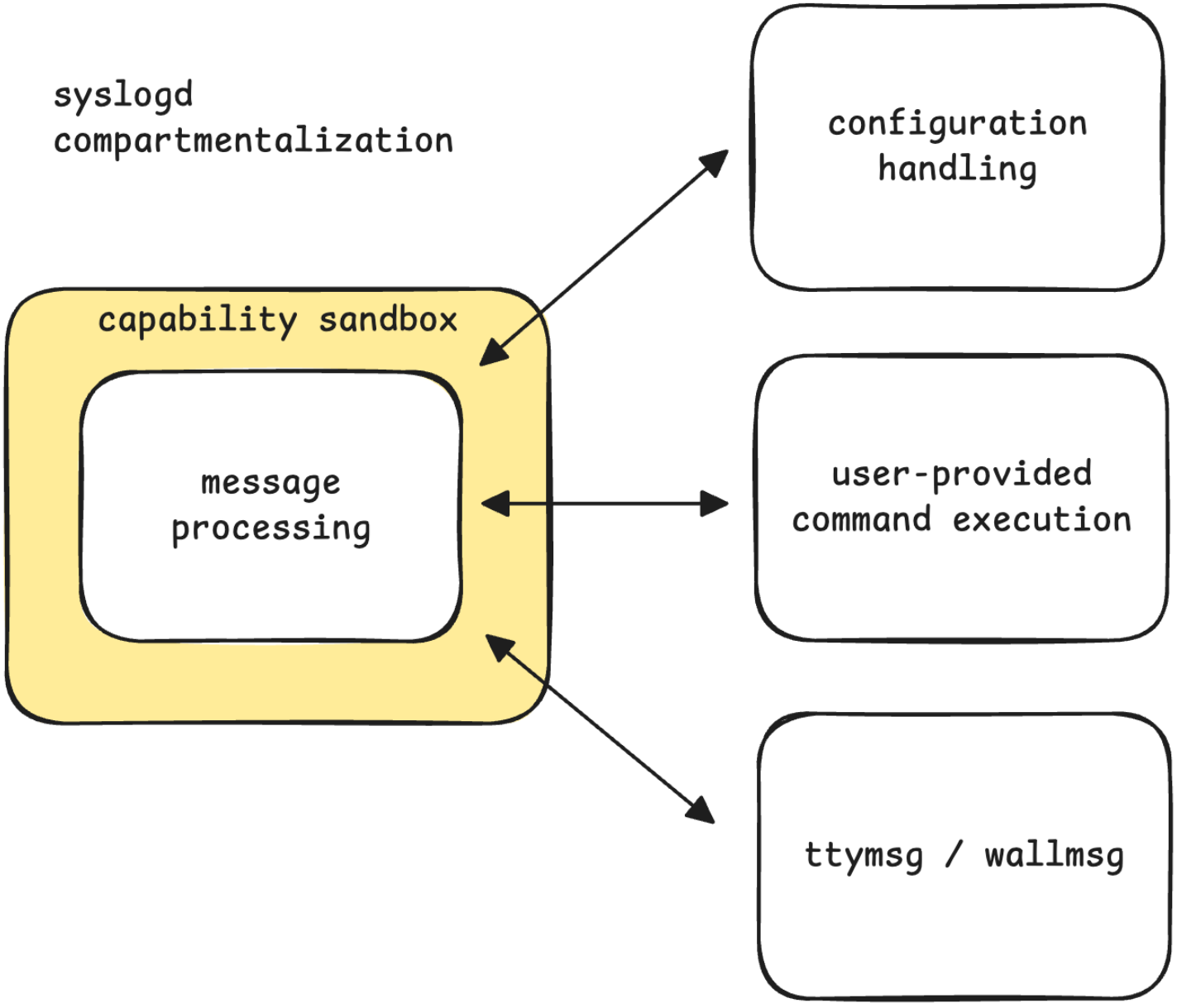
Compartmentalization

Enter compartmentalization. If the entirety of a program will not work in a sandbox, it may be possible to compartmentalize it.

Compartmentalization is the act of splitting a program up into compartments, each with their own basic purpose.

With a compartmentalized architecture, a developer can keep trusted code outside of the sandbox, but isolate insecure, or dangerous, code inside of a sandboxed compartment. If a security vulnerability is found in the dangerous code, it will be isolated.

Compartmentalization Example



Compartmentalization

Capsicum provides an intuitive interface for sandboxing specific parts of a program. At any point, a program can spawn a new child process that executes dangerous code inside of capability mode.

Interprocess communication primitives like pipes and sockets can allow data exchange without raising capability violations.

Compartmentalization Code Example

```
pid_t pid;
int pipefd[2], result;

pipe(pipefd);
/*
 * Create a child process and isolate it in a capability
 * sandbox where it can execute dangerous code.
 */
pid = fork();
if (pid == 0) {
    close(pipefd[0]);
    cap_enter();
    result = dangerous_function();
    write(pipefd[1], &result, sizeof(result));
    exit(0);
}
close(pipefd[1]);
/* Fetch result from sandboxed child. */
result = read(pipefd[0], &result, sizeof(result));
printf("Result: %d\n", result);
/* Continue normal execution in parent. */
```

Compartmentalization

Most compartments will likely need some refactoring for capability mode, but the developer can pick and choose what needs to be sandboxed.

When done right, this is less work than sandboxing the entire program, with a substantial increase in security.

Requesting Resources With libcasper(3)

Some programs were not designed to be compartmentalized. Developers of these programs could rearchitect their software, but this often requires a lot of time and resources.

Luckily, the libcasper(3) library assists developers that have complex programs where compartmentalization is not effective. Developers can use the interface provided by libcasper(3) to acquire new resources while inside of the capability sandbox.

Using a Casper Service

Casper service libraries provide a simplified interface to execute restricted operations in capability mode. To do this, the program must open a communication channel with the service before entering capability mode.

```
cap_channel_t *cap_casper, *cap_net;

/* Acquire the capability to access libcasper(3) services. */
cap_casper = cap_init();

/*
 * Use the cap_casper capability to open a communication
 * channel with the "system.net" casper service.
 */
cap_net = cap_service_open(cap_casper, "system.net");

/*
 * We do not have any more casper services to open.
 * Close the casper capability.
 */
cap_close(cap_casper);
```

Using a Casper Service

This channel can be used to request new resources from the casper service inside of capability mode.

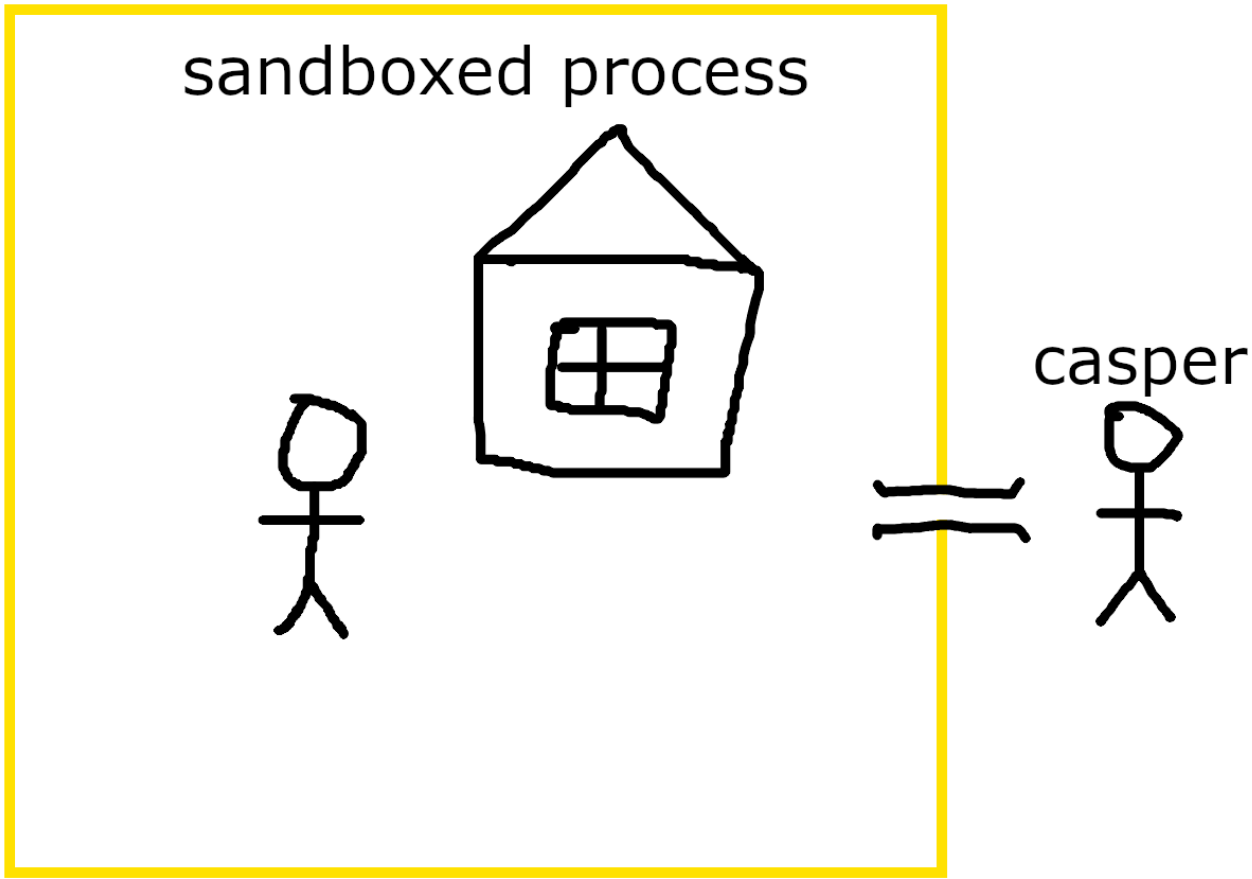
```
/*
 * Enter capability mode.
 */
cap_enter();

/*
 * Use the "cap_" variant of getaddrinfo(), provided by
 * the cap_net(3) library.
 */
cap_getaddrinfo(cap_net, "freebsd.org", "80", NULL, &res);

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

/*
 * Use the "cap_" variant of connect(), provided by
 * the cap_net(3) library.
 */
cap_connect(cap_net, s, res->ai_addr, res->ai_addrlen);
```

Casper Diagram



Channel between you and your casper process. You're essentially poking holes into your sandbox's walls.

libcasper(3) Builtins

The cap_net(3) library uses libcasper(3) to provide capability-enabled libc networking functions that would otherwise fail with ECAPMODE.

Functions that use the libcasper(3) interface are conventionally prefixed with cap_ to indicate that they succeed inside of capability mode.

There are several other casper service libraries available, similar to cap_net(3), that provide cap_-prefixed libc functions. The full list can be found on the libcasper(3) manual page.

Casper Services Under The Hood

When a sandboxed process wants a resource from a Casper process, it can use `cap_xfer_nvlist(3)`. This is what most Casper services, like `cap_net(3)`, call under the hood.

```
nvlist_t *
cap_xfer_nvlist(const cap_channel_t *chan, nvlist_t *nvl);
```

A command string and resources associated with the given command must be wrapped in an nvlist(9) before being transferred.

Communicating with the Capser Service

```
/*
 * Send the "bind" command to the casper service linked to @chan.
 */
static int
cap_bind(cap_channel_t *chan, int sockfd, const struct sockaddr *addr,
    socklen_t addrlen)
{
    nvlist_t *nvl = nvlist_create(0);
    int error;

    nvlist_add_string(nvl, "cmd", "bind");
    nvlist_add_descriptor(nvl, "sockfd", sockfd);
    nvlist_add_binary(nvl, "addr", addr, addrlen);

    nvl = cap_xfer_nvlist(chan, nvl);
    if (nvl == NULL)
        return (-1);

    error = dup2(sockfd, nvlist_get_descriptor(nvl, "sockfd"));
    nvlist_destroy(nvl);

    return (error == -1 ? -1 : 0);
}
```

Command Strings

In the previous example, `cap_bind(3)` used the "bind" command string and knew to pass a socket descriptor and sockaddr to the Casper process.

How are developers expected to know every command string and associated arguments? They're not!

On the other side of the channel, the Casper process will call its `command_func`. See next slide.

command_func

```
/*
 * The command function used by the cap_net(3) casper service
 * library.
 */
static int
net_command(const char *cmd, const nvlist_t *limits, nvlist_t *nvlin,
    nvlist_t *nvlout)
{
    if (strcmp(cmd, "bind") == 0)
        return (net_bind(limits, nvlin, nvlout));
    else if (strcmp(cmd, "connect") == 0)
        return (net_connect(limits, nvlin, nvlout));
    else if (strcmp(cmd, "gethostbyname") == 0)
        return (net_gethostbyname(limits, nvlin, nvlout));
    else if (strcmp(cmd, "gethostbyaddr") == 0)
        return (net_gethostbyaddr(limits, nvlin, nvlout));
    else if (strcmp(cmd, "getnameinfo") == 0)
        return (net_getnameinfo(limits, nvlin, nvlout));
    else if (strcmp(cmd, "getaddrinfo") == 0)
        return (net_getaddrinfo(limits, nvlin, nvlout));

    return (EINVAL);
}
```

Command Function Processing

The command function is responsible for parsing the command string and providing the appropriate response.

```
typedef int service_command_func_t(const char *cmd,
    const nvlist_t *limits, nvlist_t *nvlin,
    nvlist_t *nvlout);
```

1. `cmd`: the command string.
2. `limits`: an nvlist of limits applied to the service.
3. `nvlin`: copy of the nvlist passed into `cap_xfer_nvlist(3)` without the command string.
4. `nvlout`: the nvlist returned to `cap_xfer_nvlist(3)`.

CREATE_SERVICE(3)

That command_func can then be passed into the CREATE_SERVICE(3) macro:

```
CREATE_SERVICE(name, limit_func, command_func, flags);
```

CREATE_SERVICE(3) establishes a new Casper service that will be launched at program startup. For example, in cap_net(3):

```
CREATE_SERVICE("system.net", net_limit, net_command, 0);
```

Limiting a Casper Service

You may have noticed by now that Casper services are kind of like privilege silos.

Opening a communication channel with a Casper service for a singular purpose has the side effect of giving you access to all of the privileges offered by that service.

For this reason, most service libraries define a limits API so developers can disable functions that their program does not use.

cap_net(3) Limits

```
/*
 * Use cap_net(3)'s limitations to disable everything except
 * for resolving the address of freebsd.org on port 80.
 *
 * This assumes that the cap_net(3) service has already been
 * opened and is listening on @cap_net.
 */
cap_net_limit_t *limit;
int familylimit;

/* Allow only name resolution (cap_getaddrinfo(3)). */
limit = cap_net_limit_init(cap_net, CAPNET_NAME2ADDR);

/* Limit name resolution to "freebsd.org" on port 80. */
cap_net_limit_name2addr(limit, "freebsd.org", "80");

/* Limit name resolution to IPv4 addresses. */
familylimit = AF_INET;
cap_net_limit_name2addr_family(limit, &familylimit, 1);

/* Apply the limits to cap_net. */
cap_net_limit(limit);
```

Custom Limits

Developers may specify a `limit_func` function pointer in `CREATE_SERVICE(3)` to limit a service's interface.

When `cap_limit_set(3)` is called by a program, the provided limits are redirected to the casper service's `limit_func` where they are applied accordingly.

Limitation functions are naturally dependent on the service that they limit, so there is no clear pattern to writing them. Examples can be found in the FreeBSD source tree at `lib/libcasper/services`.

Recap: Casper Service Components

A casper service is composed of four major components:

1. Functions prefixed with `cap_` that issue commands to a casper service using `cap_xfer_nvlist(3)`.
2. A `command_func` that executes command-dependent code outside of the sandbox and returns newly acquired resources.
3. A `limit_func` that restricts what the service can be used for.
4. A `CREATE_SERVICE(3)` macro that glues the service together.

Rewind To Example Problem

You may remember this table from earlier:

Support Per Security Model				
	As root	POSIX capabilities	Pledge promises	Capsicum
Device R/W	Yes	`CAP_FOWNER`	`unix`	Device pre-open
Socket R/W	Yes	`CAP_FOWNER`	`unix`	Socket pre-open
Signals	Yes	`CAP_KILL`	`proc`	Casper sub-process
Shared memory	Yes	`CAP_IPC_OWNER`	No	Casper sub-process

In the Capsicum column, I listed "Capsicum sub-process" as the solution to "Signals" and "Shared memory". Given what you know about Casper services, you can probably imagine the steps required to create a service for these tasks.

Recap: Casper

Despite the rigidity of Casper services, they provide Capsicumized programs a necessary way to acquire resources from outside of the capability sandbox.

When used correctly, they can assist in sandboxing even the most complex programs.

Detecting Violations

When a program is placed in capability mode, it is not always obvious if it is following the rules of the sandbox.

Functions that try to open restricted resources will raise capability violations and return with errno set to ECAPMODE: Not permitted in capability mode.

Even with proper error checking, hunting down capability violations can take a lot of time. Luckily, the ktrace(2) kernel tracing utility can find violations for us.

Detecting Violations With `ktrace(2)`

Violation tracing using ktrace(2) can be started by adding two function calls at the start of any program:

```
open("ktrace.out", O_RDONLY | O_CREAT | O_TRUNC);
ktrace("ktrace.out", KTROP_SET, KTRFAC_CAPFAIL, getpid());
```

The cap_violate routine, shown next, attempts to raise every type of violation that ktrace(2) can capture. It is not important to understand what the routine is doing, just that it raises capability violations.

ktrace(2) Example

```
open("ktrace.out", O_RDONLY | O_CREAT | O_TRUNC);
ktrace("ktrace.out", KTROP_SET, KTRFAC_CAPFAIL, getpid());

cap_rights_init(&rights, CAP_READ);
caph_rights_limit(STDERR_FILENO, &rights);
write(STDERR_FILENO, &val, sizeof(val));

cap_rights_set(&rights, CAP_WRITE);
caph_rights_limit(STDERR_FILENO, &rights);

kinf.kf_structsize = sizeof(struct kinfo_file);
fcntl(STDIN_FILENO, F_KINFO, &kinf);

socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

addr.sin_family = AF_INET;
addr.sin_port = htons(5000);
addr.sin_addr.s_addr = INADDR_ANY;
bind(socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP),
    (const struct sockaddr *)&addr, sizeof(addr));
sendto(fd, NULL, 0, 0, (const struct sockaddr *)&addr, sizeof(addr));

kill(getppid(), SIGCONT);

openat(AT_FDCWD, "/", O_RDONLY);

CPU_SET(0, &cpuset_mask);
cpuset_setaffinity(CPU_LEVEL_WHICH, CPU_WHICH_PID, getppid(),
    sizeof(cpuset_mask), &cpuset_mask);
```

ktrace(2) Dump Results

```
# ./cap_violate
# kdump
  1915 cap_violate CAP   operation requires CAP_WRITE, descriptor holds CAP_READ
  1915 cap_violate CAP   attempt to increase capabilities from CAP_READ to CAP_READ,CAP_WRITE
  1915 cap_violate CAP   system call not allowed: fcntl, cmd: F_KINFO
  1915 cap_violate CAP   socket: protocol not allowed: IPPROTO_ICMP
  1915 cap_violate CAP   system call not allowed: bind
  1915 cap_violate CAP   sendto: restricted address lookup: struct sockaddr { AF_INET, 0.0.0.0:5000 }
  1915 cap_violate CAP   kill: signal delivery not allowed: SIGCONT
  1915 cap_violate CAP   openat: restricted VFS lookup: AT_FDCWD
  1915 cap_violate CAP   cpuset_setaffinity: restricted cpuset operation
```

Recap: Tracing

Violation tracing is just another tool in the developer's toolbox. It only takes a few seconds to run a program under ktrace(2) and the result is almost always a decent starting point for sandboxing your program using Capsicum.

Overall Recap

Capability mode and all other security mechanisms mentioned in this talk were designed to make programs safer.

Capability mode provides robust security by isolating a program from the rest of the system.

If capability mode is properly integrated, a developer can rest assured that their program is safer than it was before introducing Capsicum.

Thank you for your attention!

ask questions

Sources and Extra Resources

- https://cdaemon.com/posts/capsicum
- https://www.cl.cam.ac.uk/research/security/capsicum
- https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf
- https://wiki.freebsd.org/Capsicum