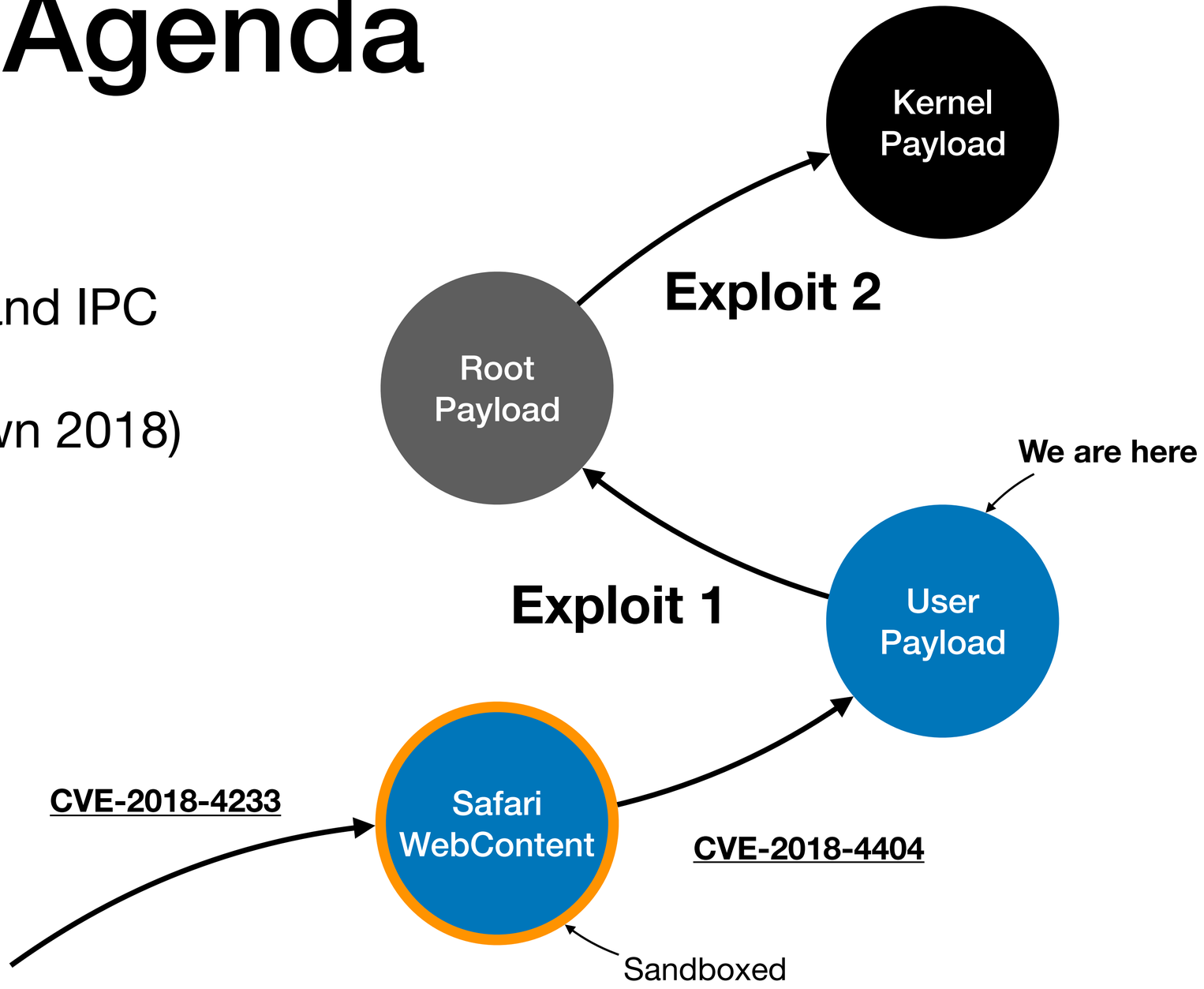
macOS IPC MitM

Samuel Groß (@5aelo)

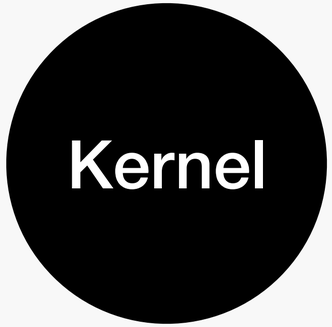
Agenda

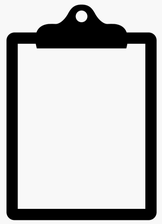
1. Apple's OS Architecture and IPC
2. CVE-2018-4237 (Pwn2Own 2018)
3. Exploit 1: user -> root
4. Exploit 2: root -> kernel
5. Demo



"Classic" OS Design

Kernel:
- Manages all resources
- Performs access control
- Runs fully privileged

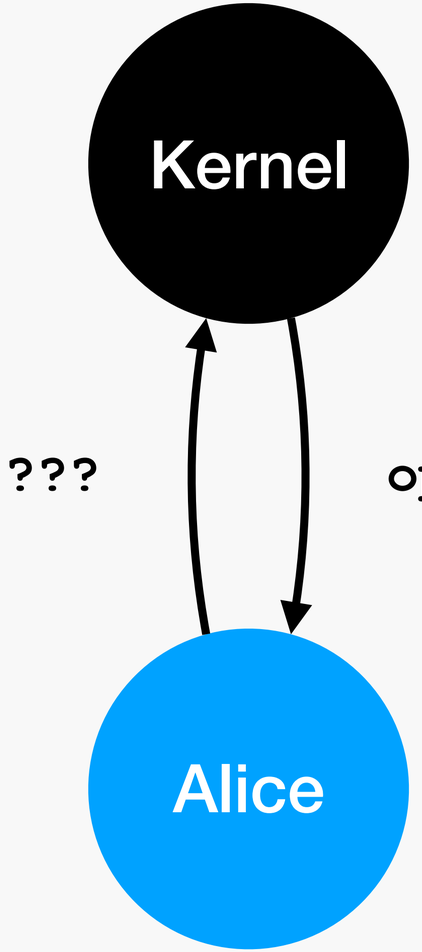




`/home/bob/secrets.txt rw-rw----`

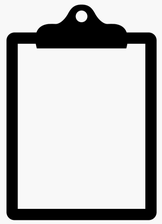
"Classic" OS Design

Kernel:
- Manages all resources
- Performs access control
- Runs fully privileged



???

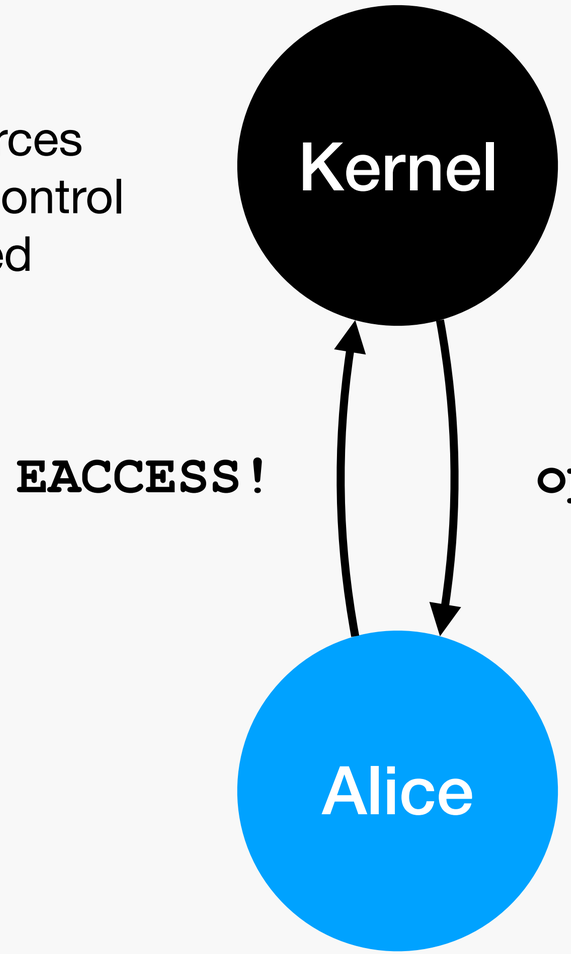
```
open("/home/bob/secrets.txt", O_RDONLY)
```



```
/home/bob/secrets.txt rw-rw----
```

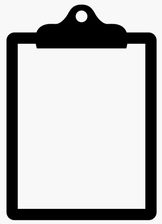
"Classic" OS Design

Kernel:
- Manages all resources
- Performs access control
- Runs fully privileged



Compare request bits (r--) with access bits for that user (---)

```
open("/home/bob/secrets.txt", O_RDONLY)
```



```
/home/bob/secrets.txt rw-rw----
```

Userspace Resources

Wanted: resource management in **userspace**

- Cloud documents, contacts, UI events, clipboard, preferences, keychain, ... are all userspace "resources"

Benefits of managing things in userspace:

- Userspace code probably easier to write than kernel code
- Access to memory safe languages (e.g. Swift on macOS)
- Small, restricted services that can be sandboxed to only have access to the resources they need
- Synchronized access easy: (single-threaded) daemon handling requests

Preferences

- Preferences = persistent, per application key:value pairs
- "Resource" managed in userspace, by cfprefsd
- Programatic access: CF Preferences
- CLI access: `defaults`

```
> defaults write net.saelo.hax foo bar
> defaults read net.saelo.hax
{
    foo = bar;
}
> plutil -p ~/Library/\
Preferences/net.saelo.hax.plist
{
  "foo" => "bar"
}
```

Preferences

Goal: write/update a preference

Preferences

cfprefsd:

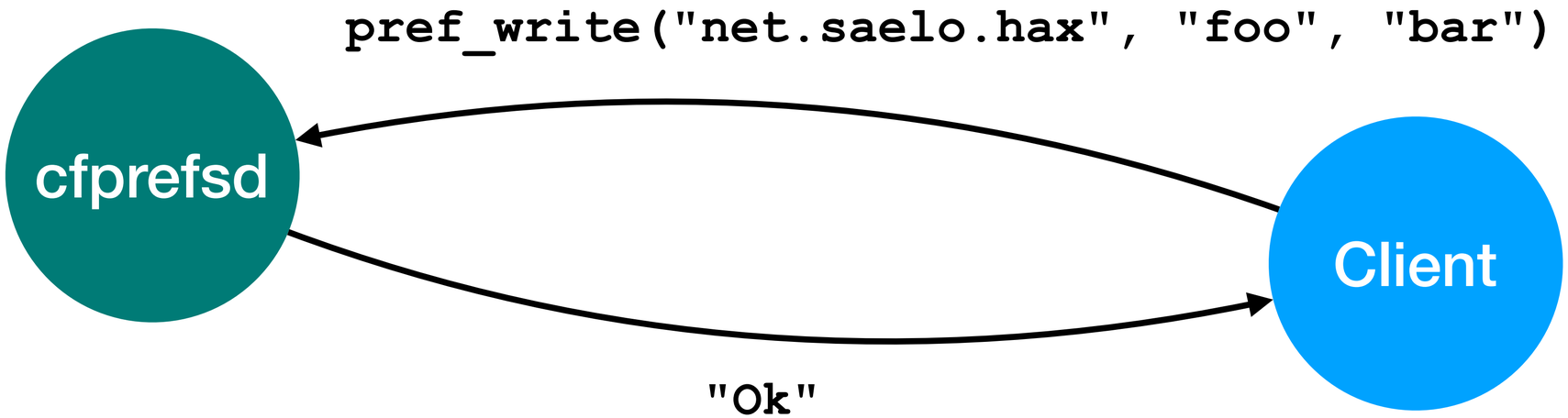
- Manages one resource
- Performs access control
  - E.g. denies access to sandboxed clients
- Runs as user, can be sandboxed

Goal: write/update a preference

Preferences

cfprefsd:

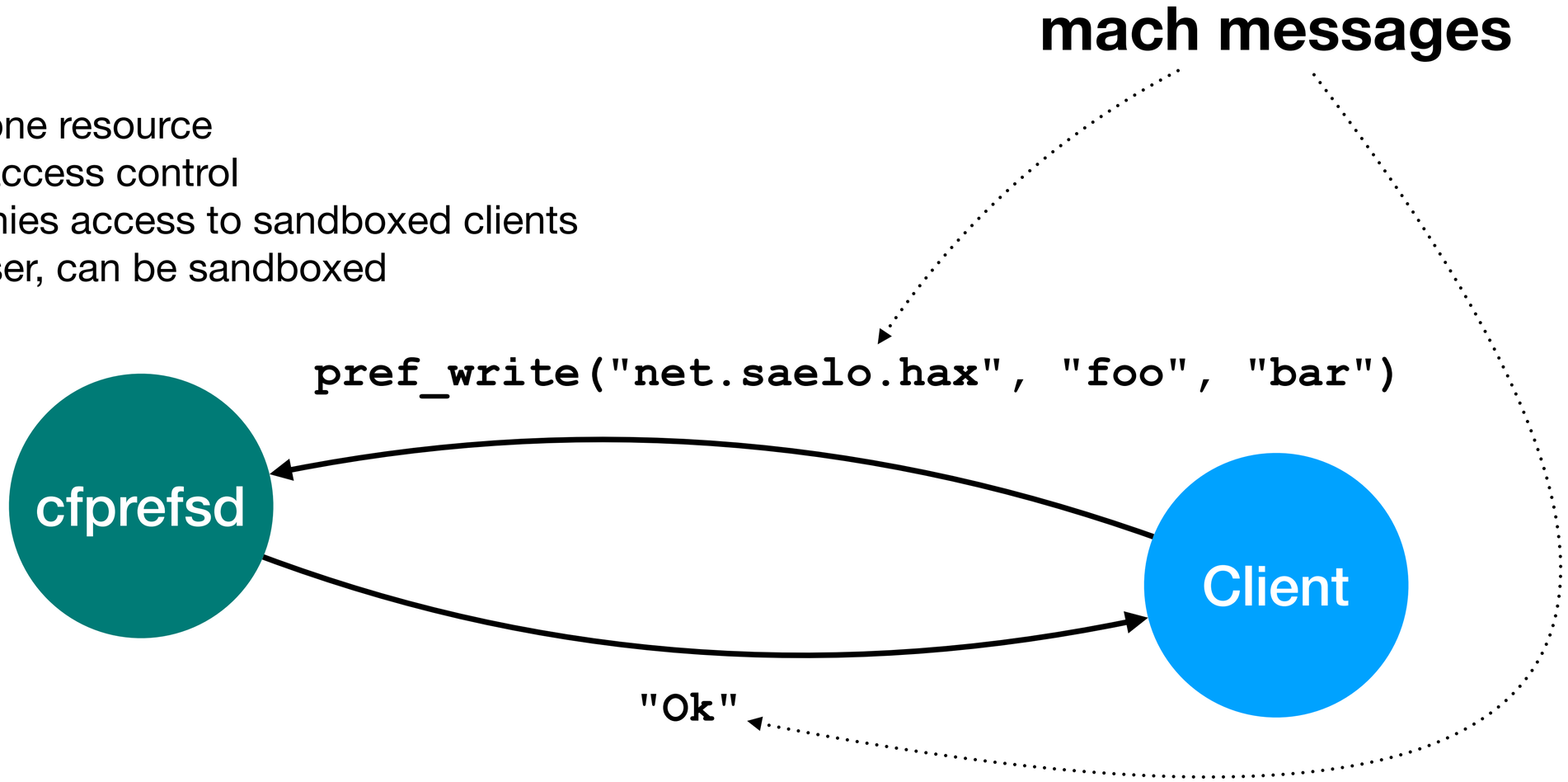
- Manages one resource
- Performs access control
  - E.g. denies access to sandboxed clients
- Runs as user, can be sandboxed



Preferences

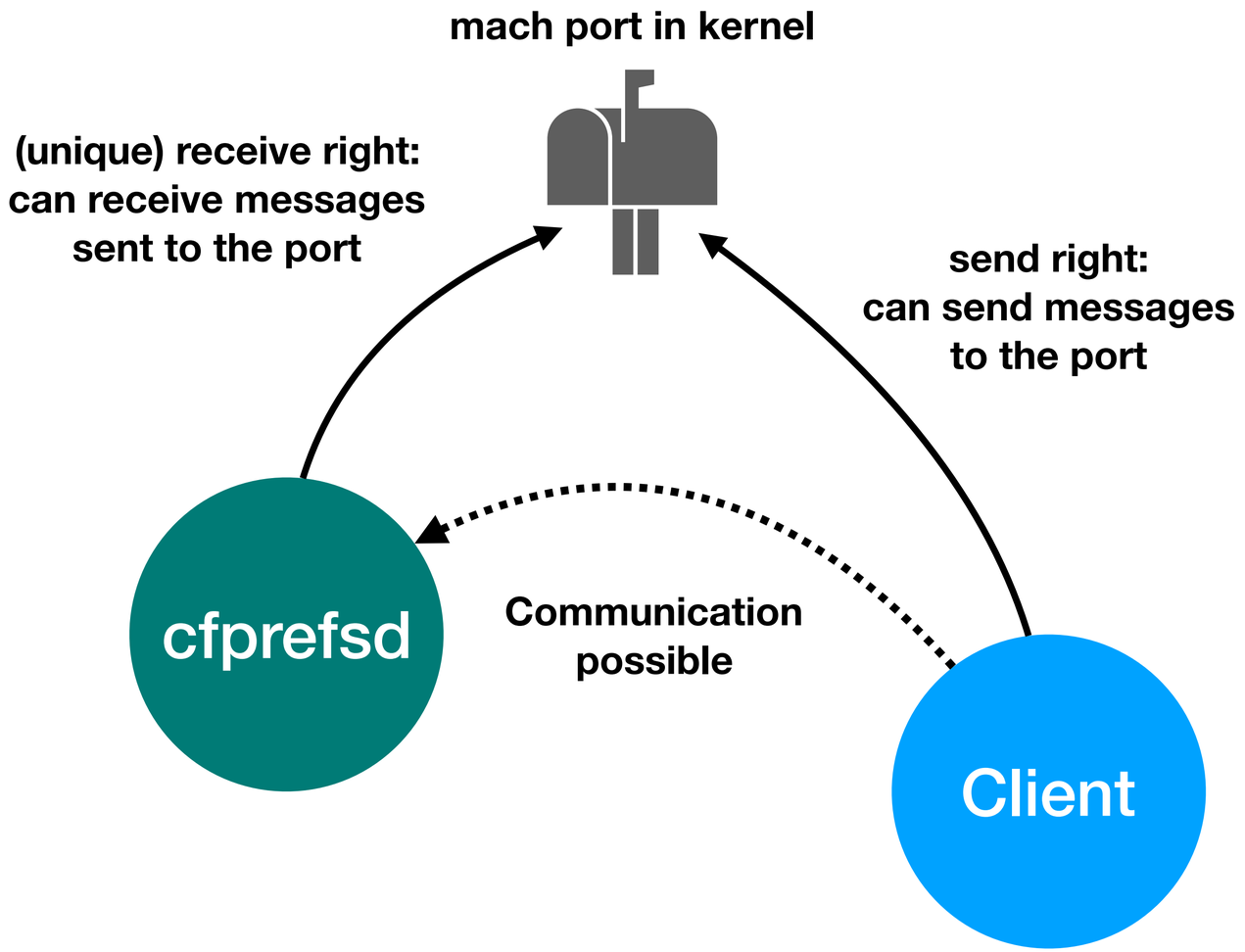
cfprefsd:

- Manages one resource
- Performs access control
  - E.g. denies access to sandboxed clients
- Runs as user, can be sandboxed

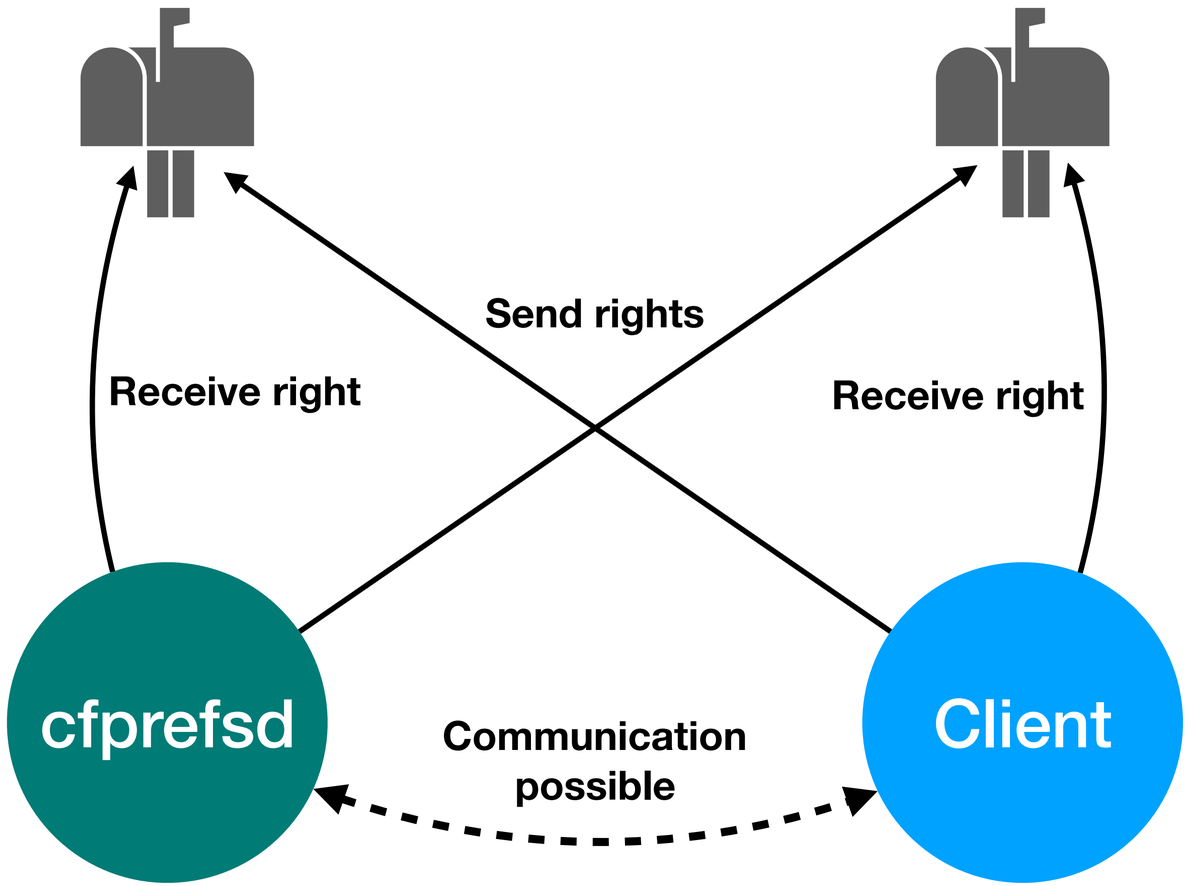


Mach Messaging

- Fundamental IPC mechanism in Darwin: mach messages
  - Based on mach ports: unidirectional, mailbox-style IPC
- Sender needs a *send right* to a mach port for which the service process owns the *receive right*
- *Send-once right* to another mach port can be attached to a message to receive a reply



XPC



- IPC protocol built on top of mach messages
- Supports sending key:value dictionaries
- XPC connection consists of two mach ports: one for sending, one for receiving
- Reply ports (send-once right attached to message) still used when reply expected (e.g. RPC)

Service Management

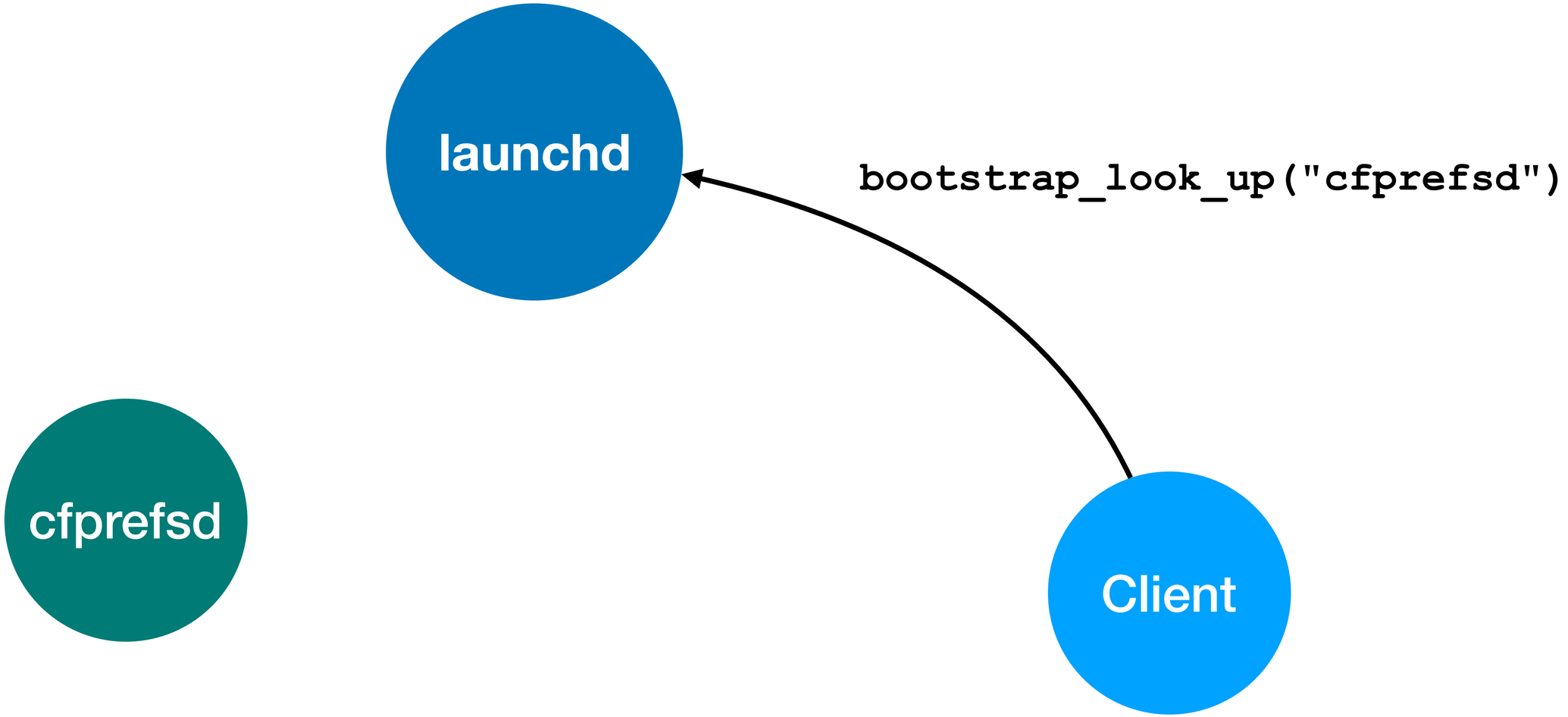
Question: how can client "find" cfprefsd?

Service Management

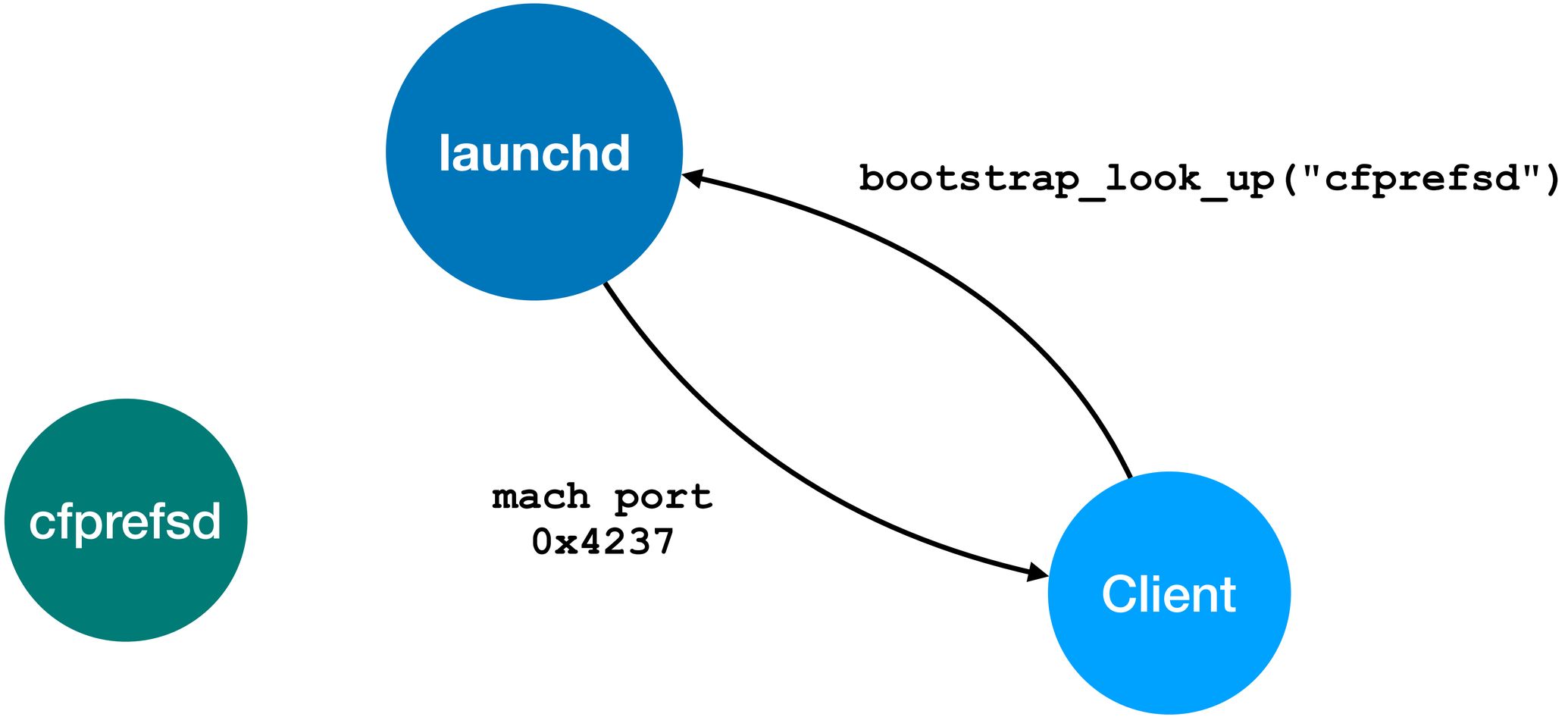
launchd:

- Init process (pid 1)
- Manages IPC services
  - Every service registers with launchd
- Highly privileged

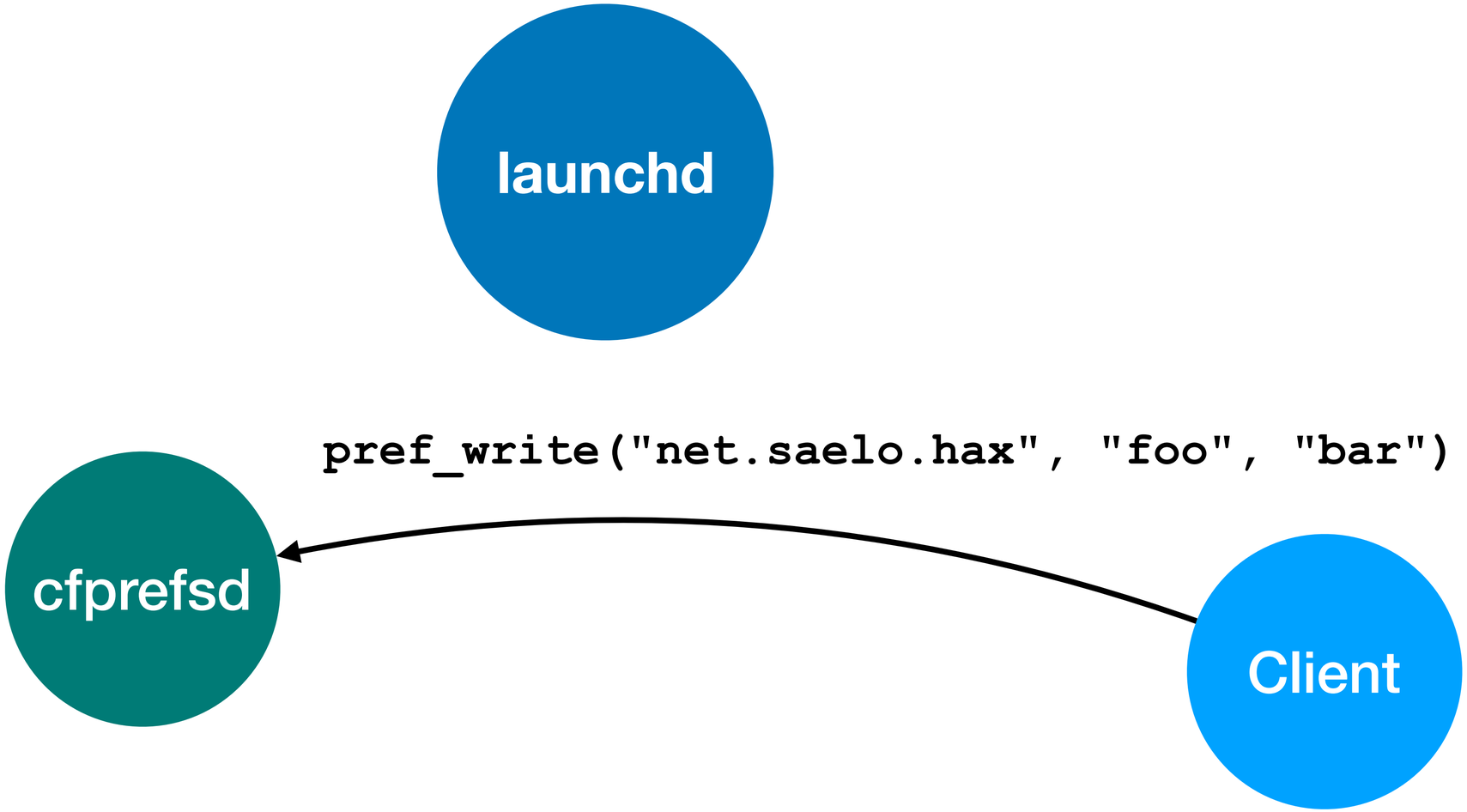
Service Management



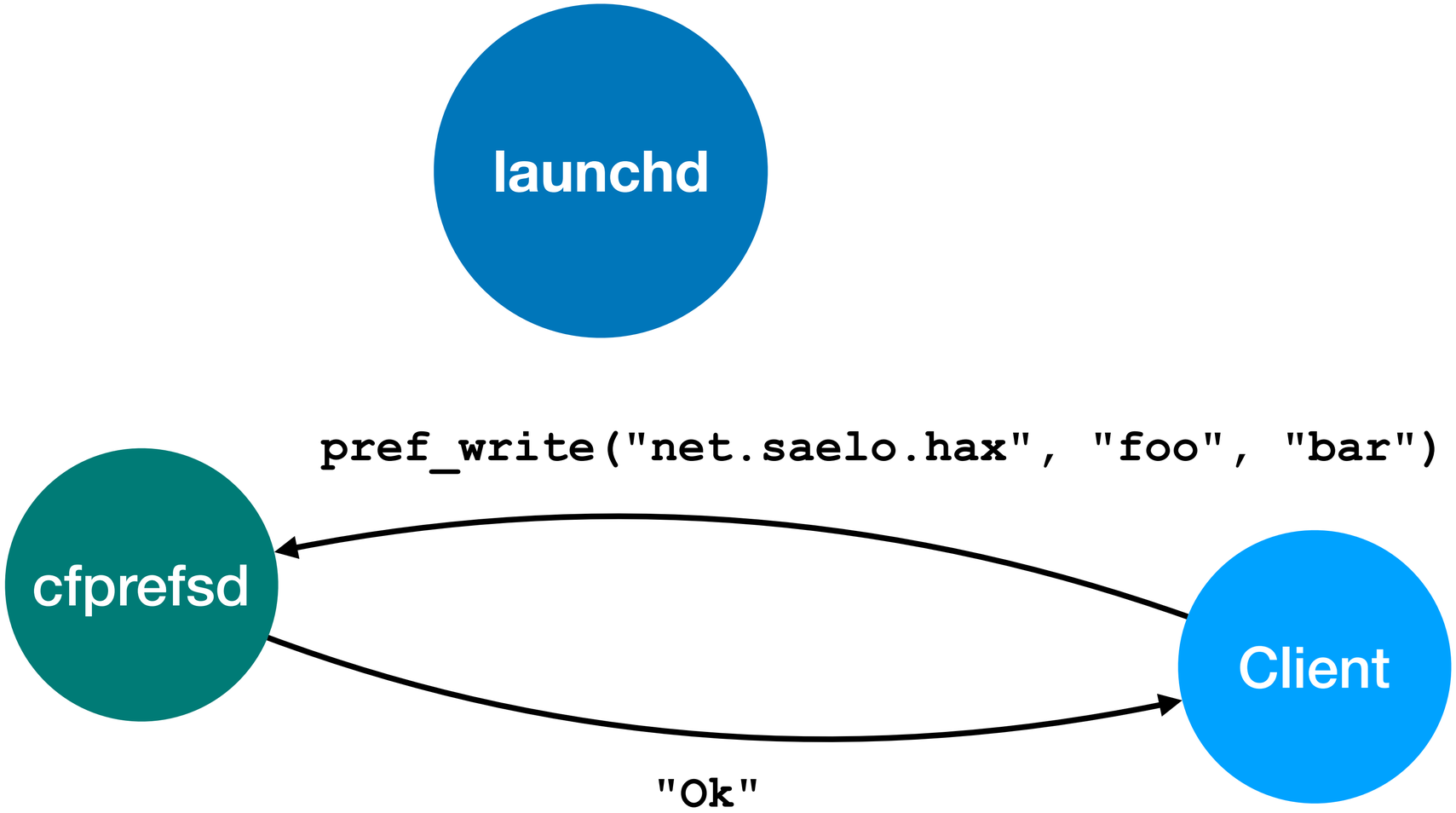
Service Management



Service Management



Service Management



Task Special Ports

- Problem: how can a process communicate with launchd in the first place?
- Solution: one of the task special ports, the *bootstrap port*, is connected to launchd

  => Messages sent to the bootstrap port will arrive in launchd

```
typedef int task_special_port_t;

#define TASK_KERNEL_PORT            1
#define TASK_HOST_PORT              2
#define TASK_NAME_PORT              3
#define TASK_BOOTSTRAP_PORT         4
#define TASK_SEATBELT_PORT          7
#define TASK_ACCESS_PORT            9
#define TASK_DEBUG_CONTROL_PORT     10
#define TASK_RESOURCE_NOTIFY_PORT   11
```

task_set_special_port

```
/*
 * Set one of the special ports
 * associated with the target task.
 */
routine task_set_special_port(
      task           : task_t;
      which_port     : int;
      special_port   : mach_port_t
);
```

- `task_set_special_port` API allows overwriting special ports, including the bootstrap port
- Overwritten bootstrap port not restored during `fork()` or `execve()`
- 🤔
- Spawn privileged child process (e.g. a setuid binary) and intercept IPC?

  => CVE-2018-4237

task_set_special_port

```
/*
 * Set one of the special ports
 * associated with the target task.
 */
routine task_set_special_port(
      task          : task_t;
      which_port    : int;
      special_port  : mach_port_t
);
```

- `task_set_special_port` API allows overwriting special ports, including the bootstrap port
- Overwritten bootstrap port not restored during `fork()` or `execve()`
- 🤔
- Spawn privileged child process (e.g. a setuid binary) and intercept IPC?

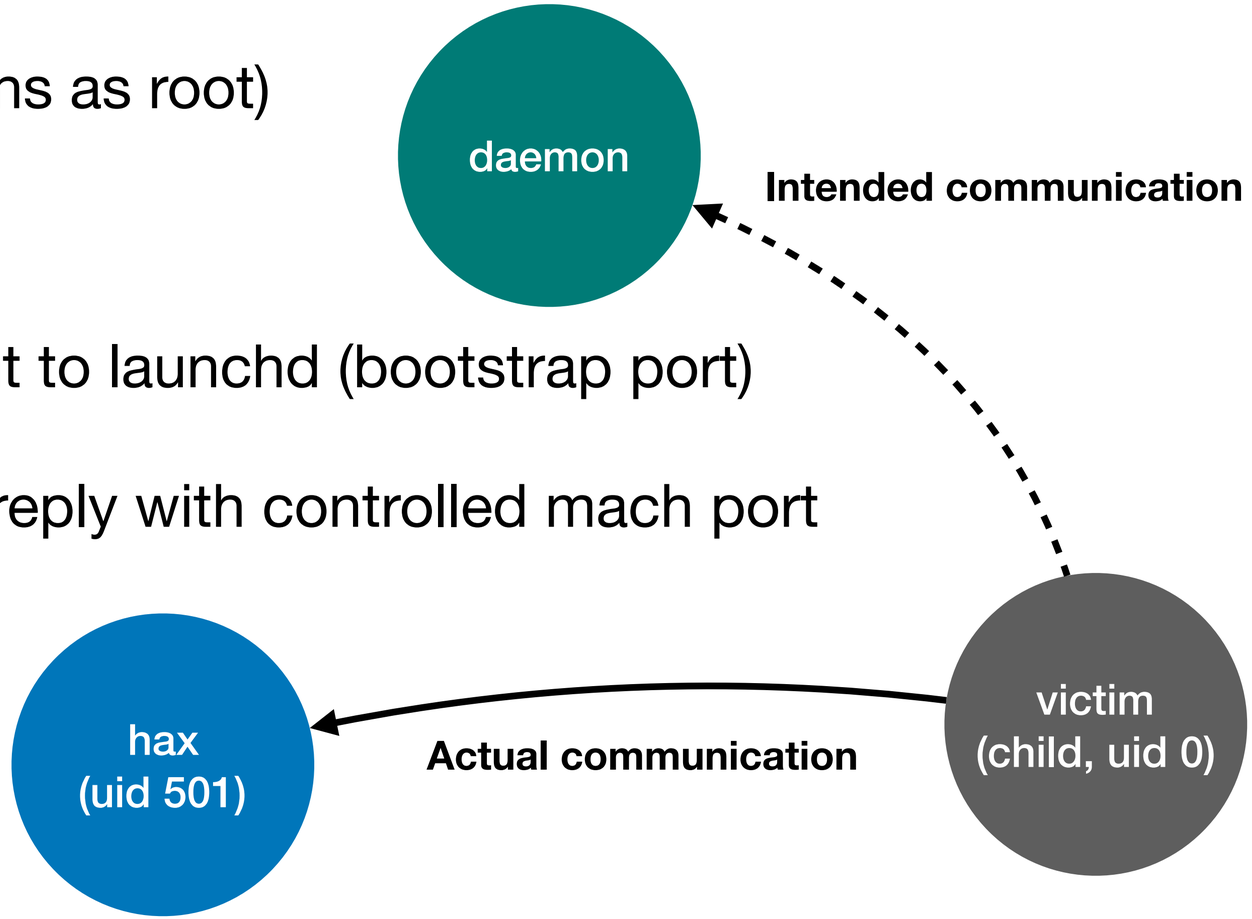
=> CVE-2018-4237

* Fun sidenote: exploit basically described in https://robert.sesek.com/2014/1/changes_to_xnu_mach_ipc.html

CVE-2018-4237

- Security bug: child process can be more privileged than parent
  - Due to setuid bit being set (child runs as root)
  - Or due to entitlements
- Primitive: can intercept messages sent to launchd (bootstrap port)
- Idea: intercept endpoint lookups and reply with controlled mach port

  => IPC man-in-the-middle



Normal Sudo

launchd

```
> sudo whoami
Password:
root
```

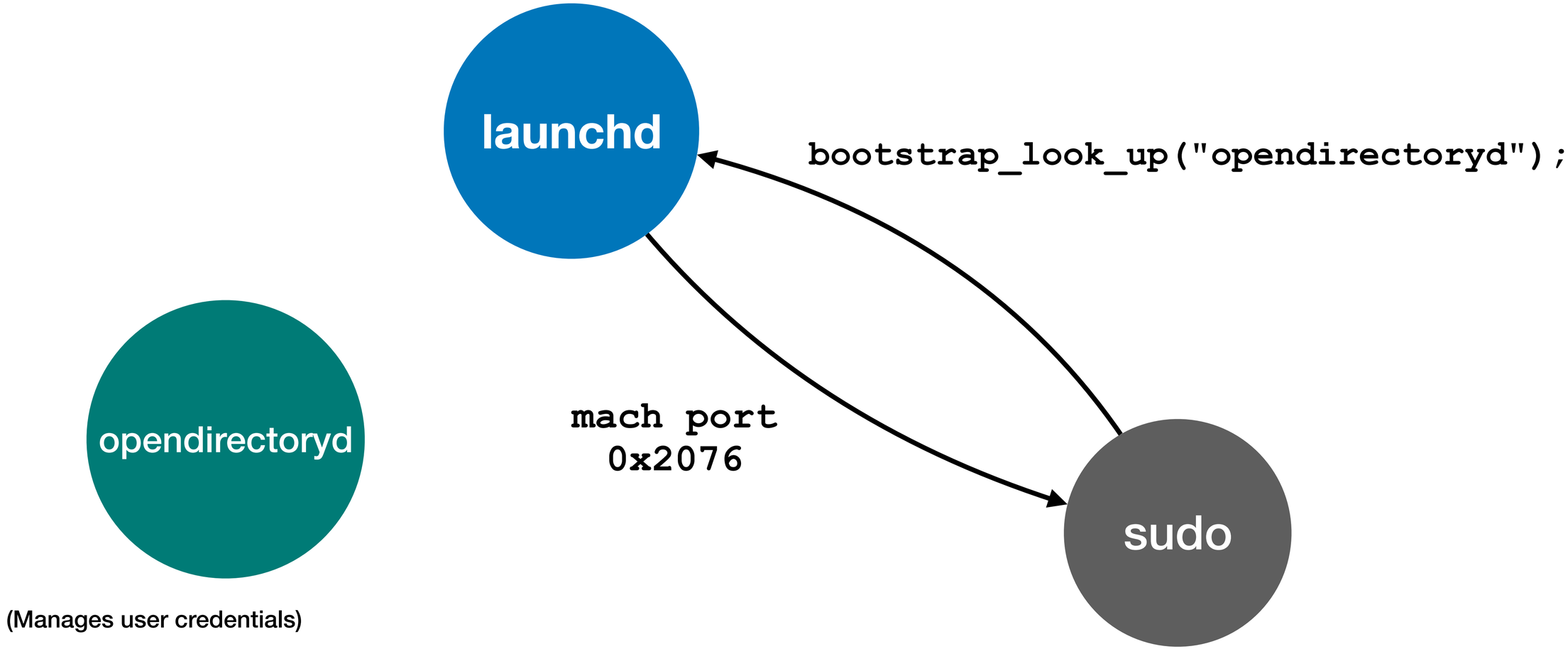
opendirectoryd

(Manages user credentials)

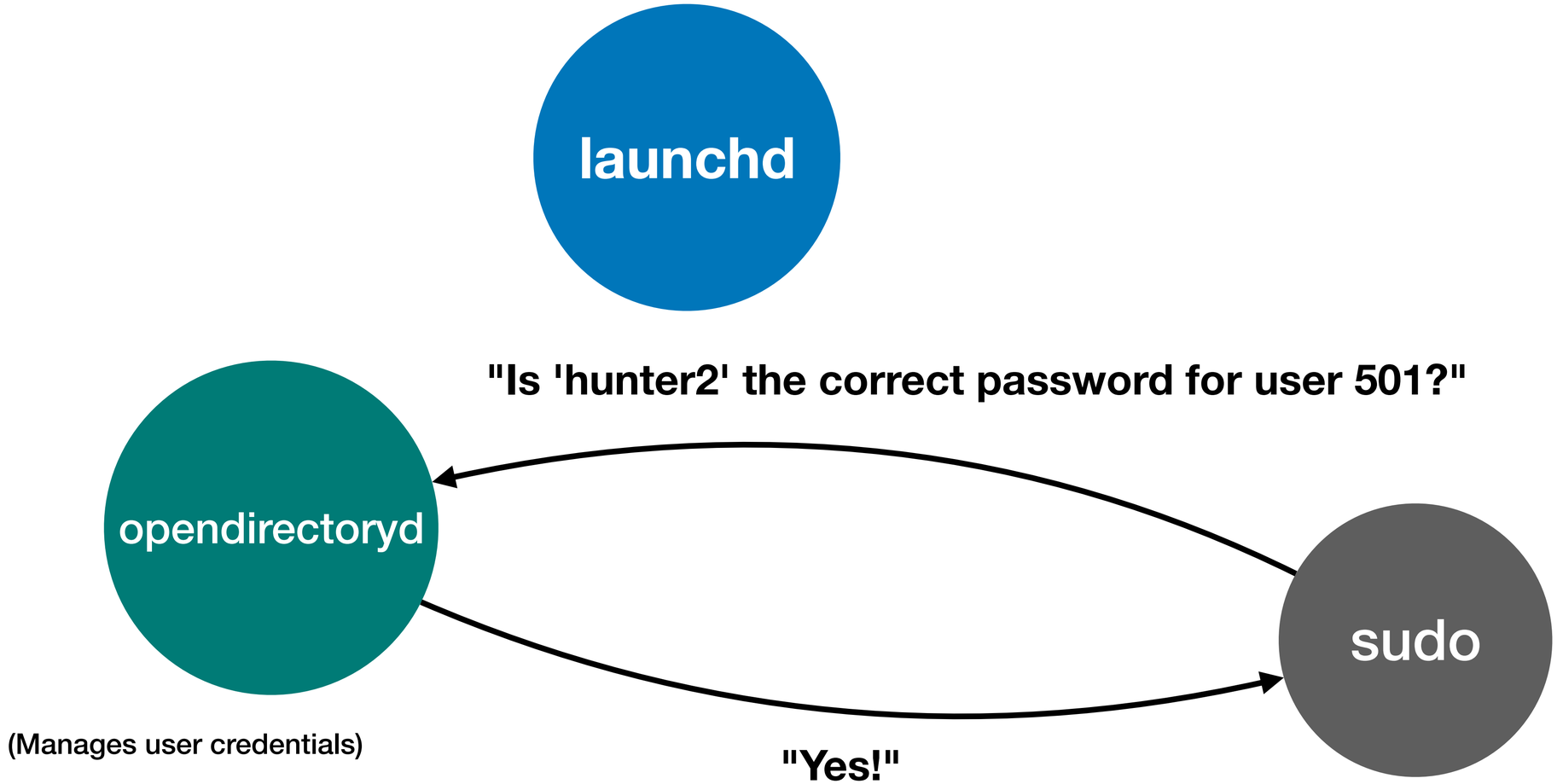
What happens here?

sudo

Normal Sudo


launchd
bootstrap_look_up("opendirectoryd");
mach port
0x2076
opendirectoryd
sudo
(Manages user credentials)

Normal Sudo


launchd
"Is 'hunter2' the correct password for user 501?"
opendirectoryd
sudo
(Manages user credentials)
"Yes!"

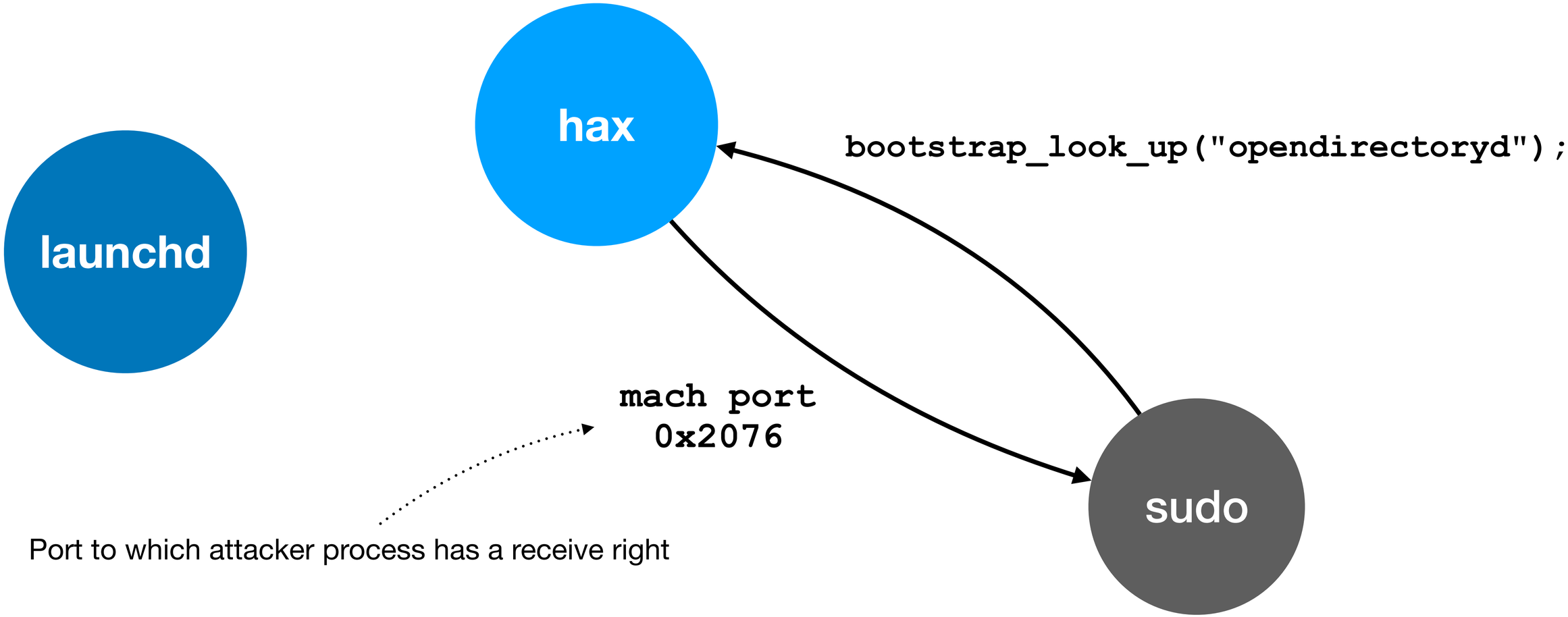
Exploit 1 - Idea

- MitM XPC connection between sudo (child process) and opendirectoryd
- Send arbitrary password to sudo over stdin

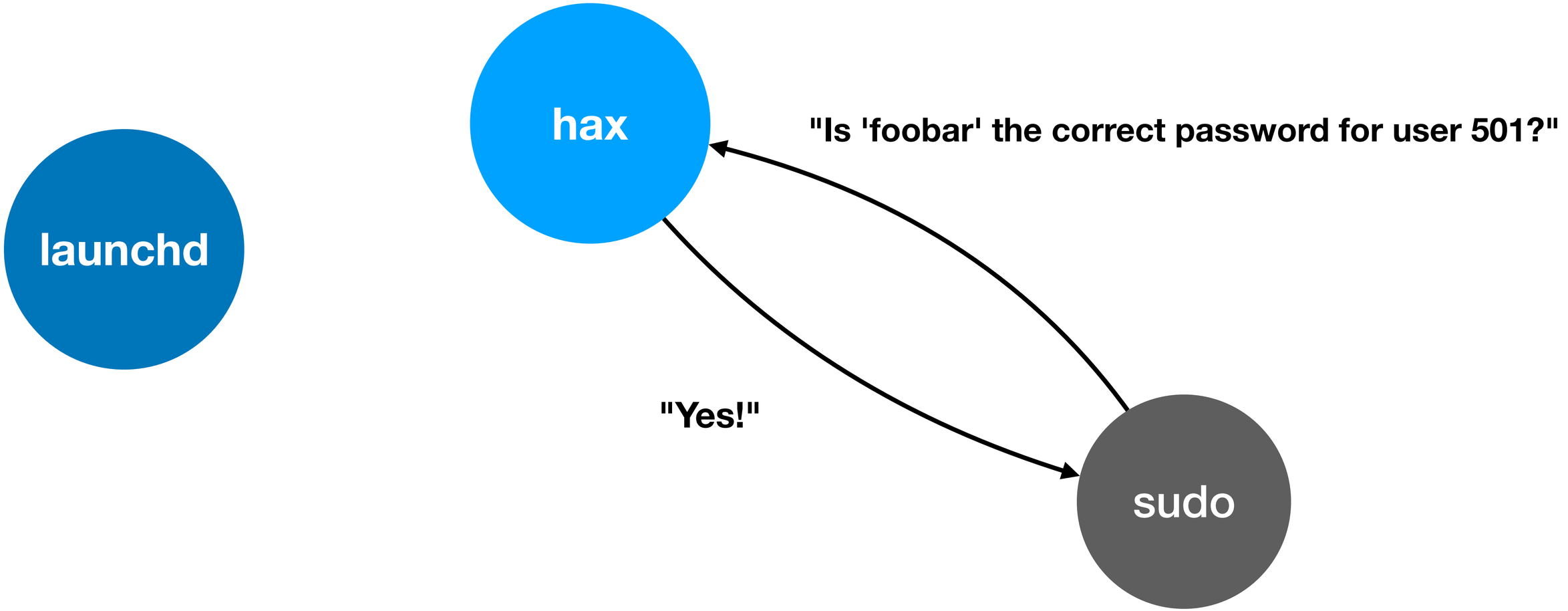
    => sudo will send password to opendirectoryd for verification

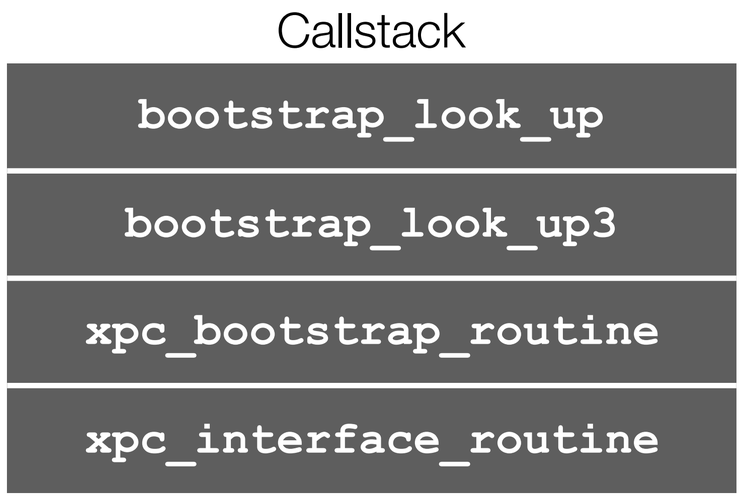
- Intercept reply from opendirectoryd to indicate that password is valid

Exploit 1 - Idea



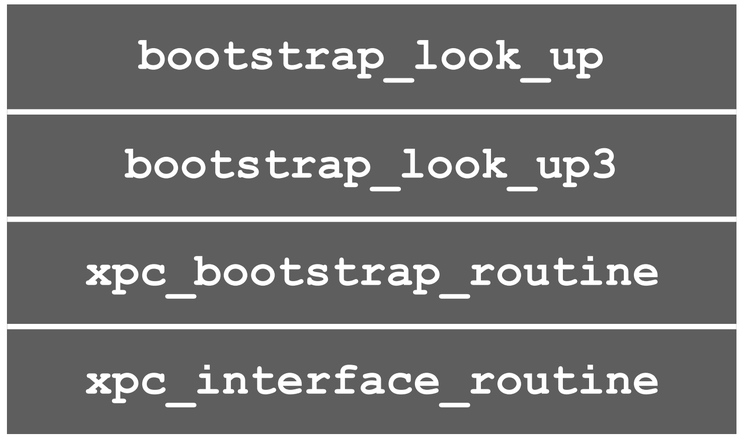
Exploit 1 - Idea




Callstack
bootstrap_look_up
bootstrap_look_up3
xpc_bootstrap_routine
xpc_interface_routine

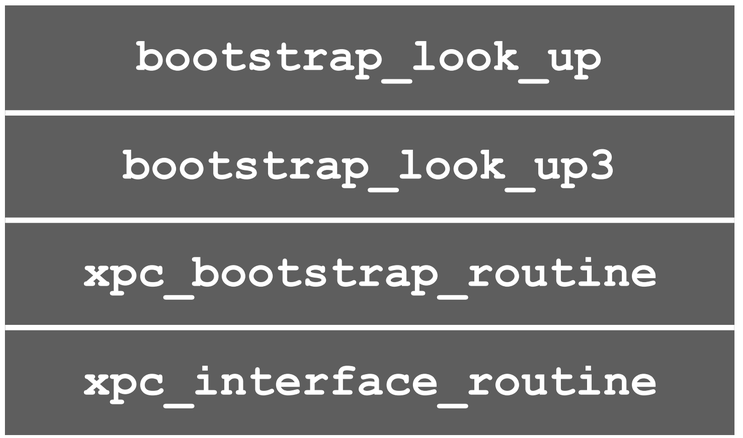
```
int xpc_interface_routine(int subsystem, int routine,
    xpc_dictionary_t msg, xpc_dictionary_t* out)
{
  ...;
  xpc_dictionary_set_uint64(msg, "subsystem", subsystem);
  xpc_dictionary_set_uint64(msg, "routine", routine);
  r = xpc_pipe_routine(msg, &response);
  if (!r) {
    xpc_dictionary_get_audit_token(response, &token);
    if (token.pid != 1 || token.euid) {
      return 118;
    }
    ...;
}
```

Callstack



```
int xpc_interface_routine(int subsystem, int routine,
    xpc_dictionary_t msg, xpc_dictionary_t* out)
{
  ...;
  xpc_dictionary_set_uint64(msg, "subsystem", subsystem);
  xpc_dictionary_set_uint64(msg, "routine", routine);
  r = xpc_pipe_routine(msg, &response);
  if (!r) {
    xpc_dictionary_get_audit_token(response, &token);
    if (token.pid != 1 || token.euid) {
      return 118;
    }
    ...;
}
```

Callstack



Problem

Problem: victim (libxpc) verifies that reply came from launchd (pid == 1, uid == 0)

Solution:

1. Register endpoint, e.g. "net.saelo.hax", with launchd via `bootstrap_register`
2. Intercept lookup requests from victim to launchd and
    1. Change endpoint name to "net.saelo.hax"
    2. Leave original reply port intact!
    3. Forward to launchd

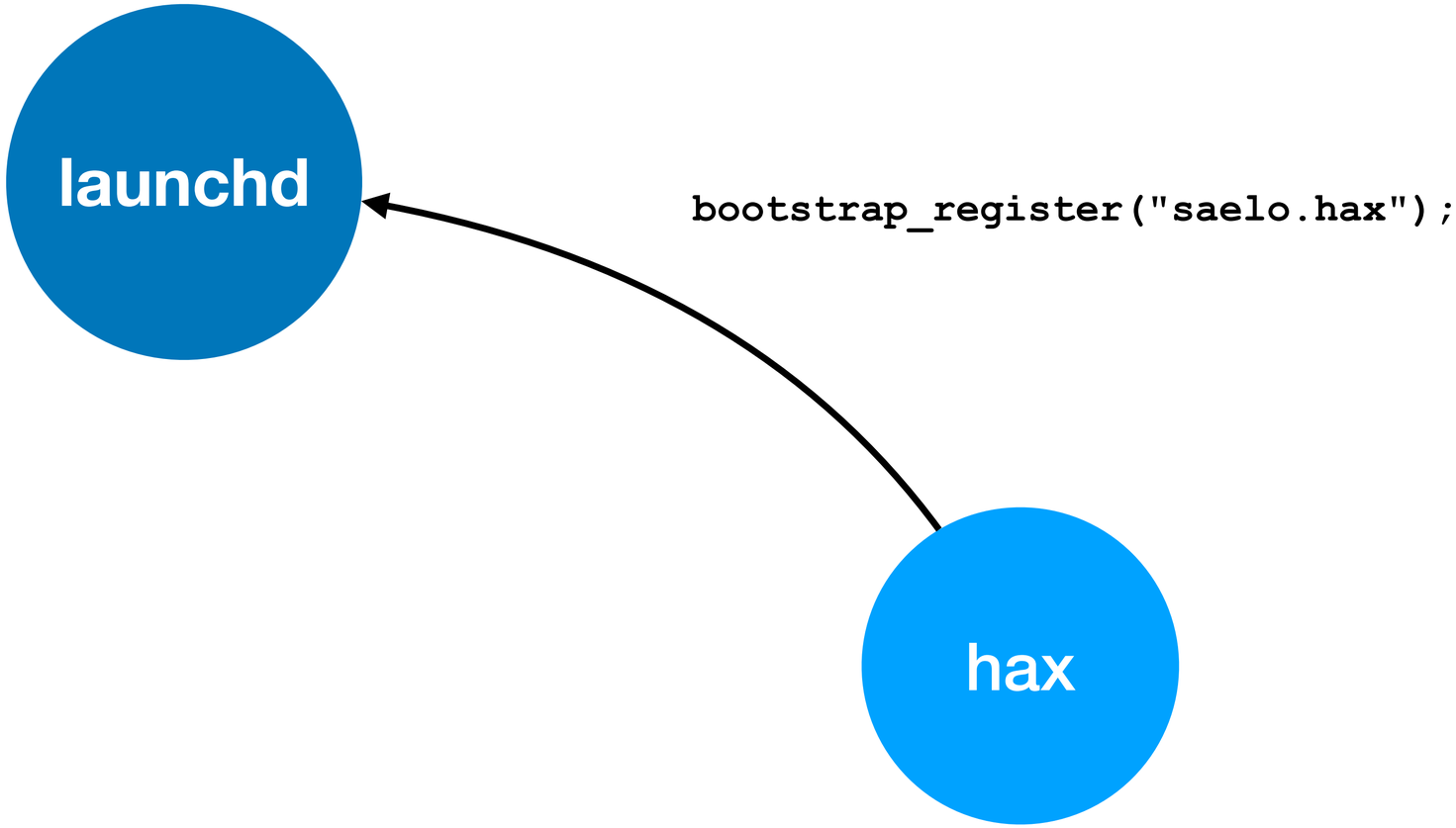
XPC Packet

- **Sender Port**
- **Reply Port (!)**
- **Message ID**
- **Serialized Message**

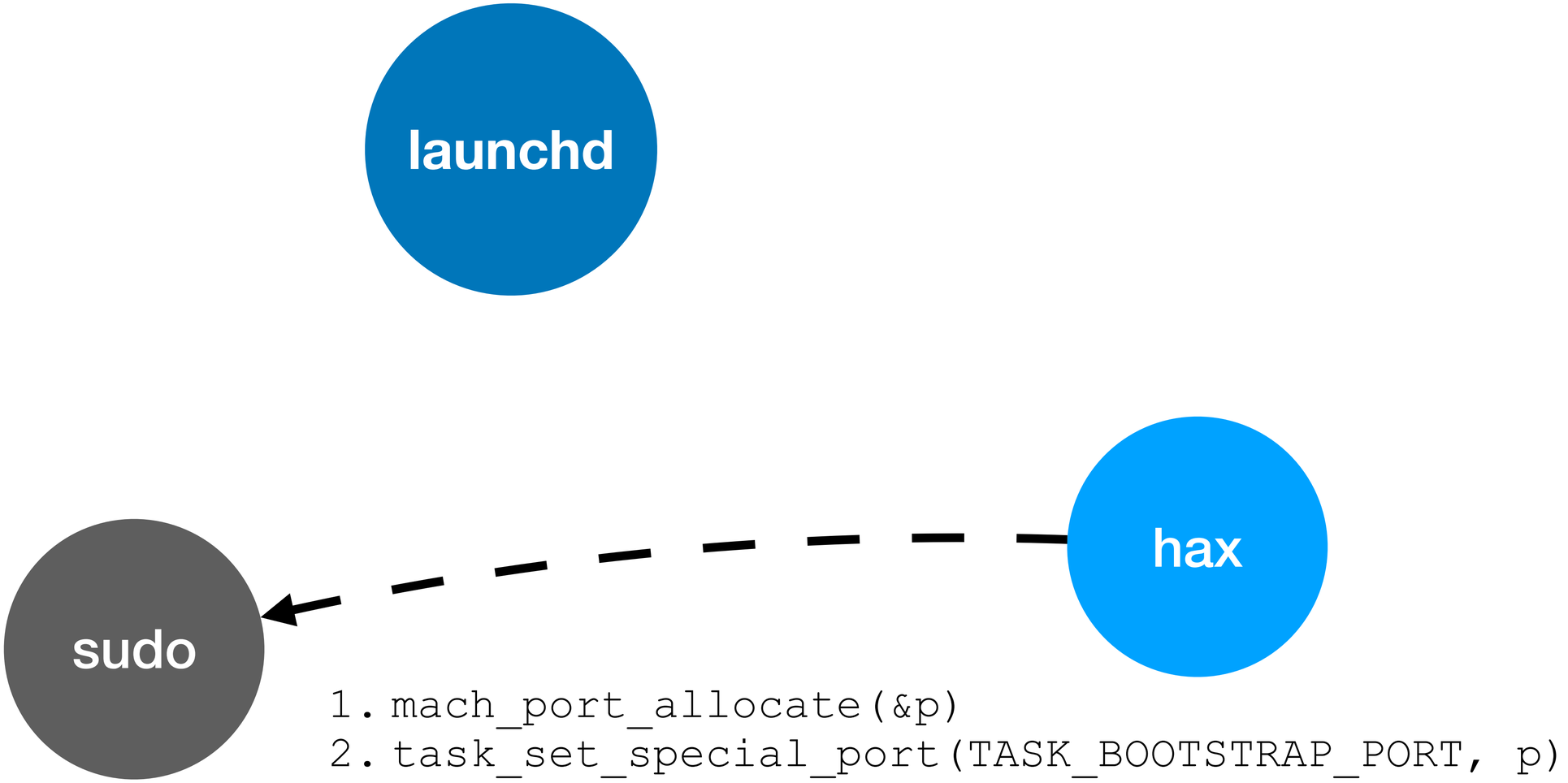
=> launchd will reply directly to victim process with controlled IPC port!

Passwordless Sudo

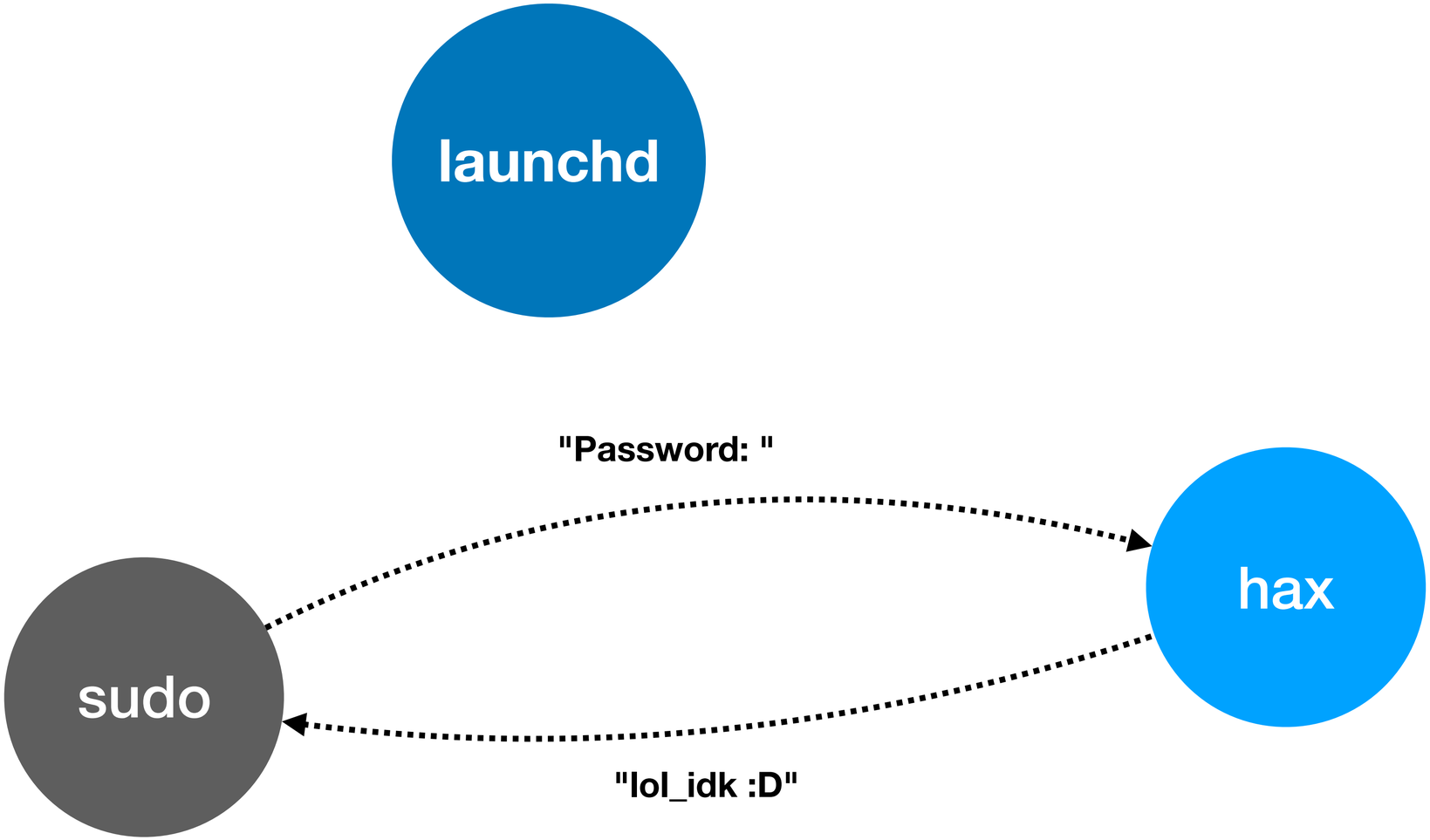
Passwordless Sudo



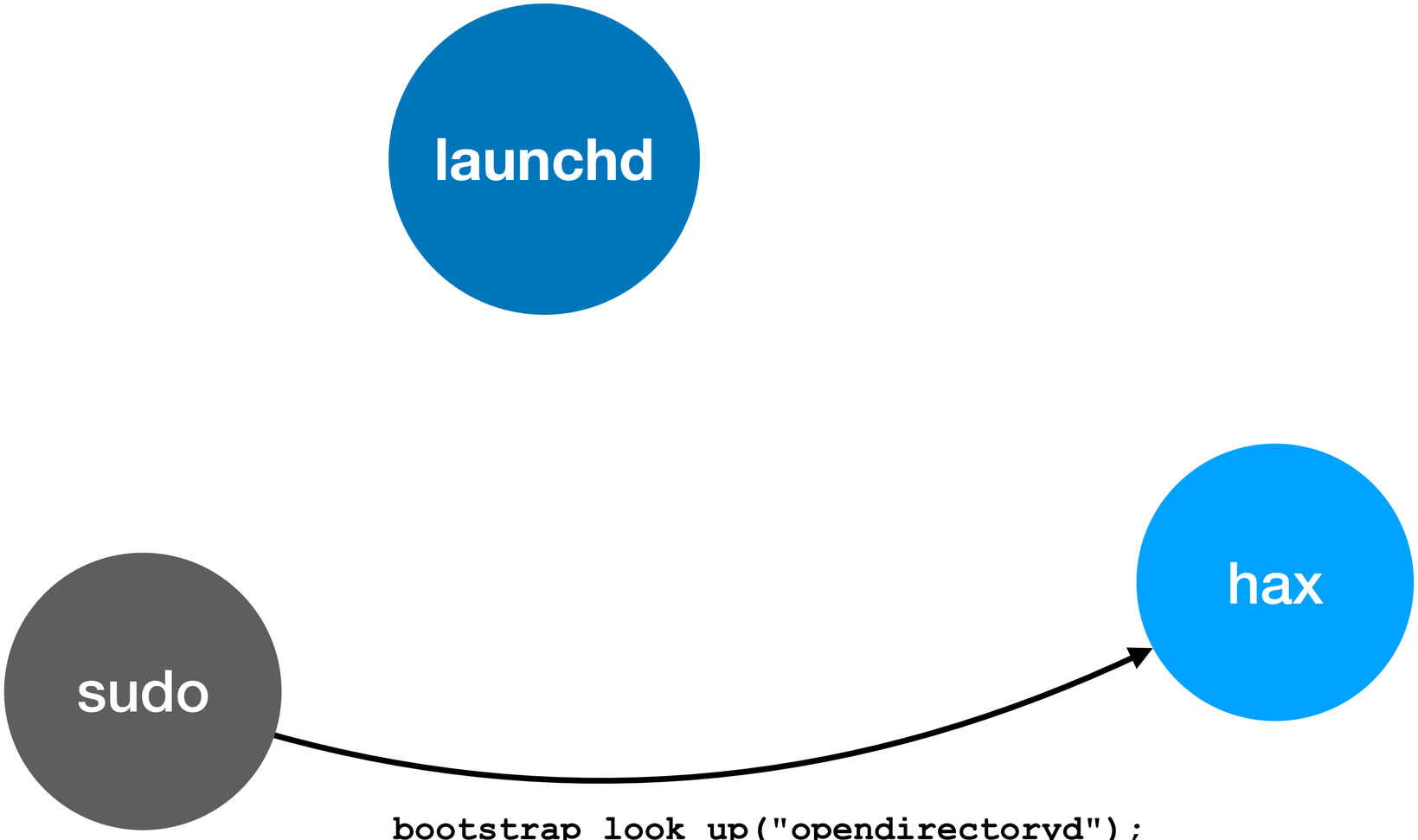
Passwordless Sudo



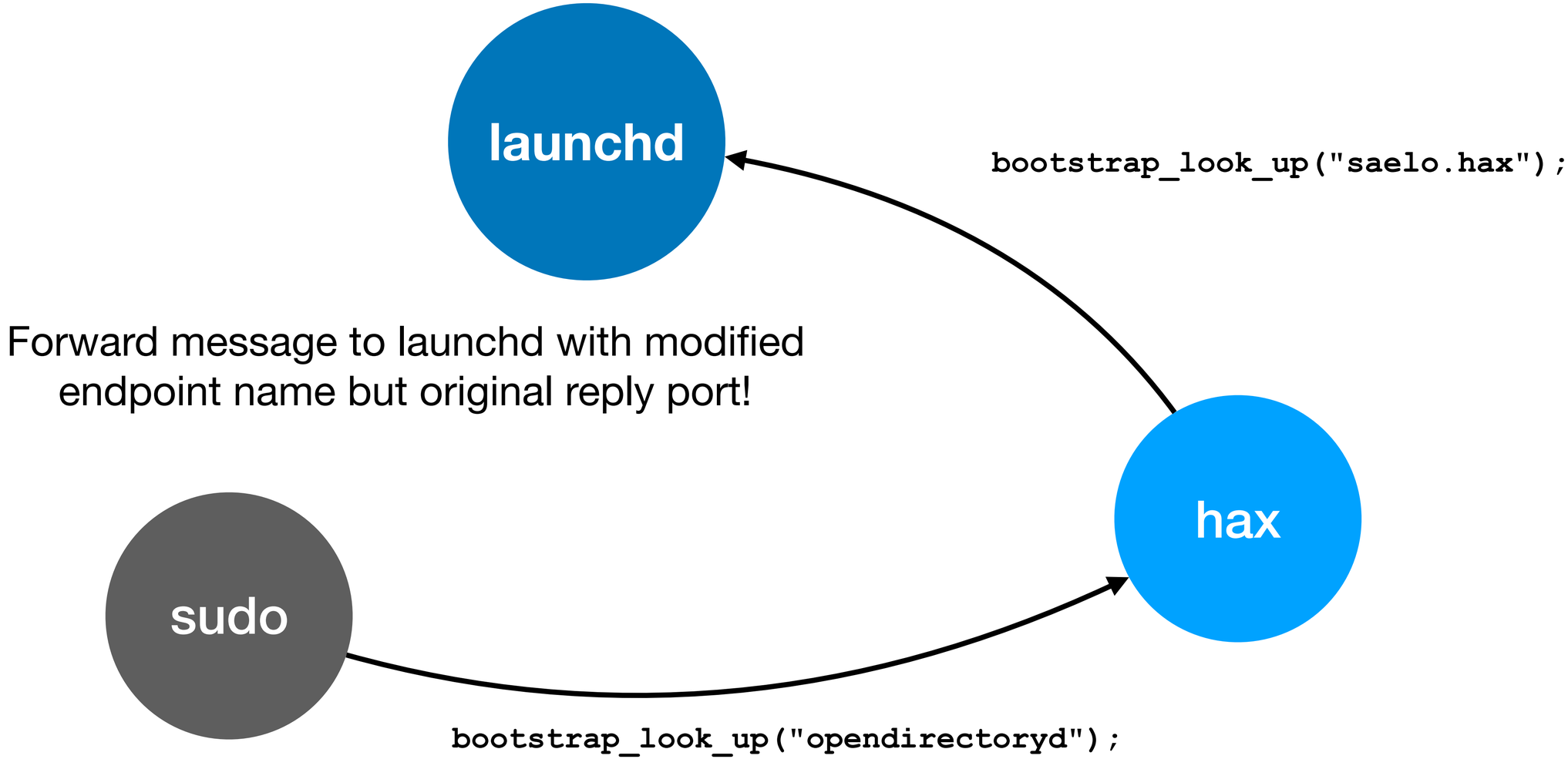
Passwordless Sudo



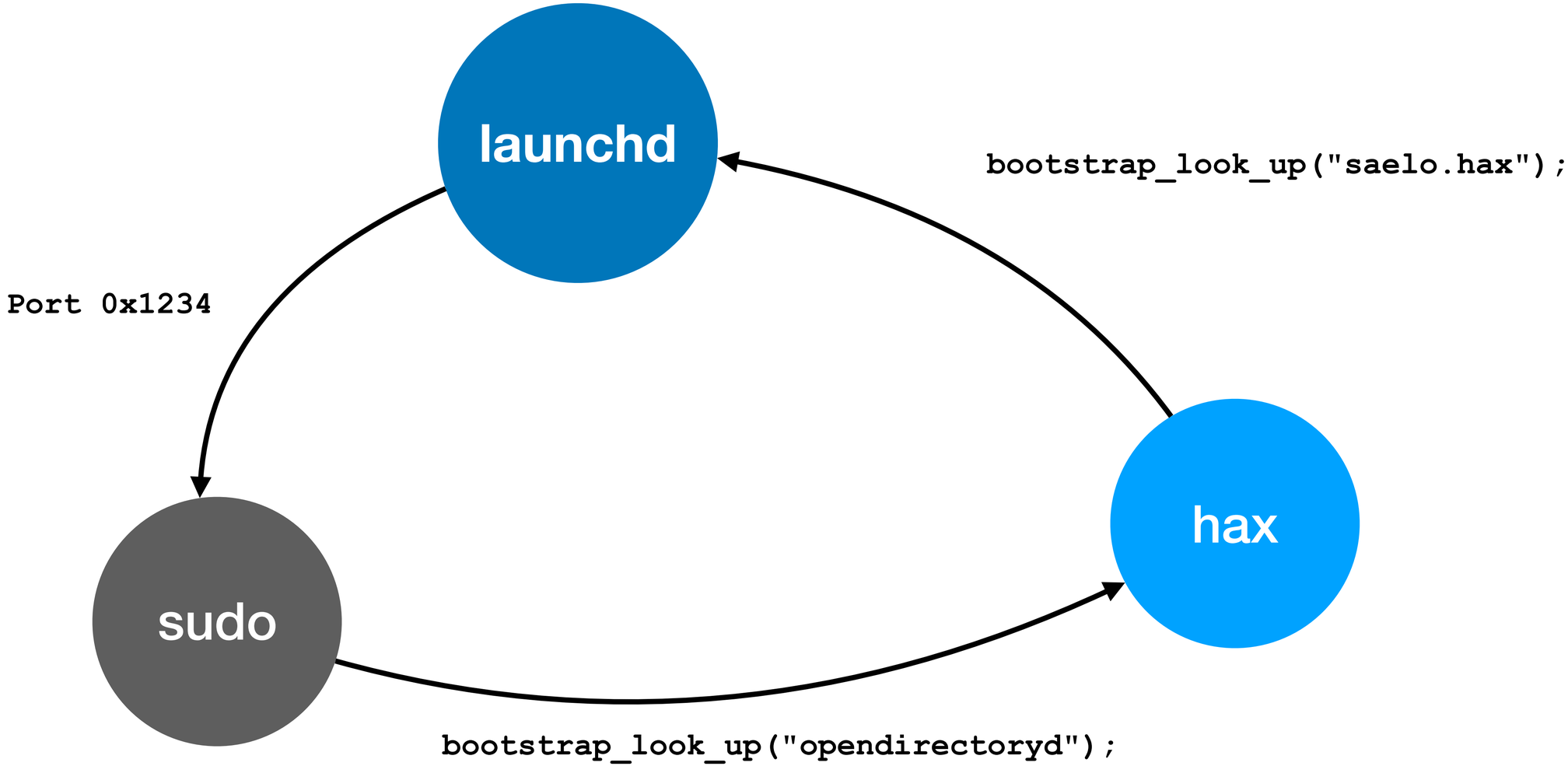
Passwordless Sudo



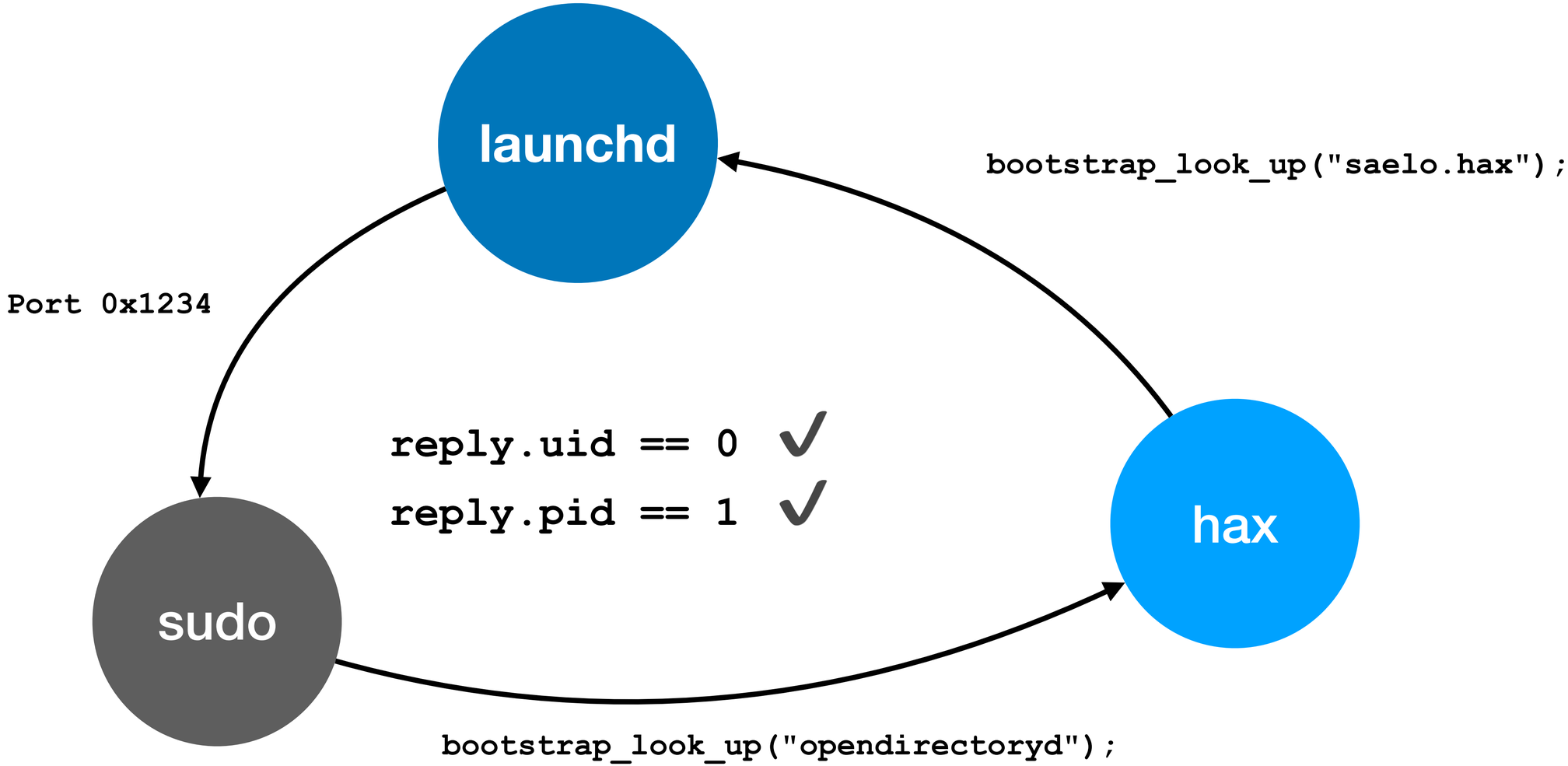
Passwordless Sudo



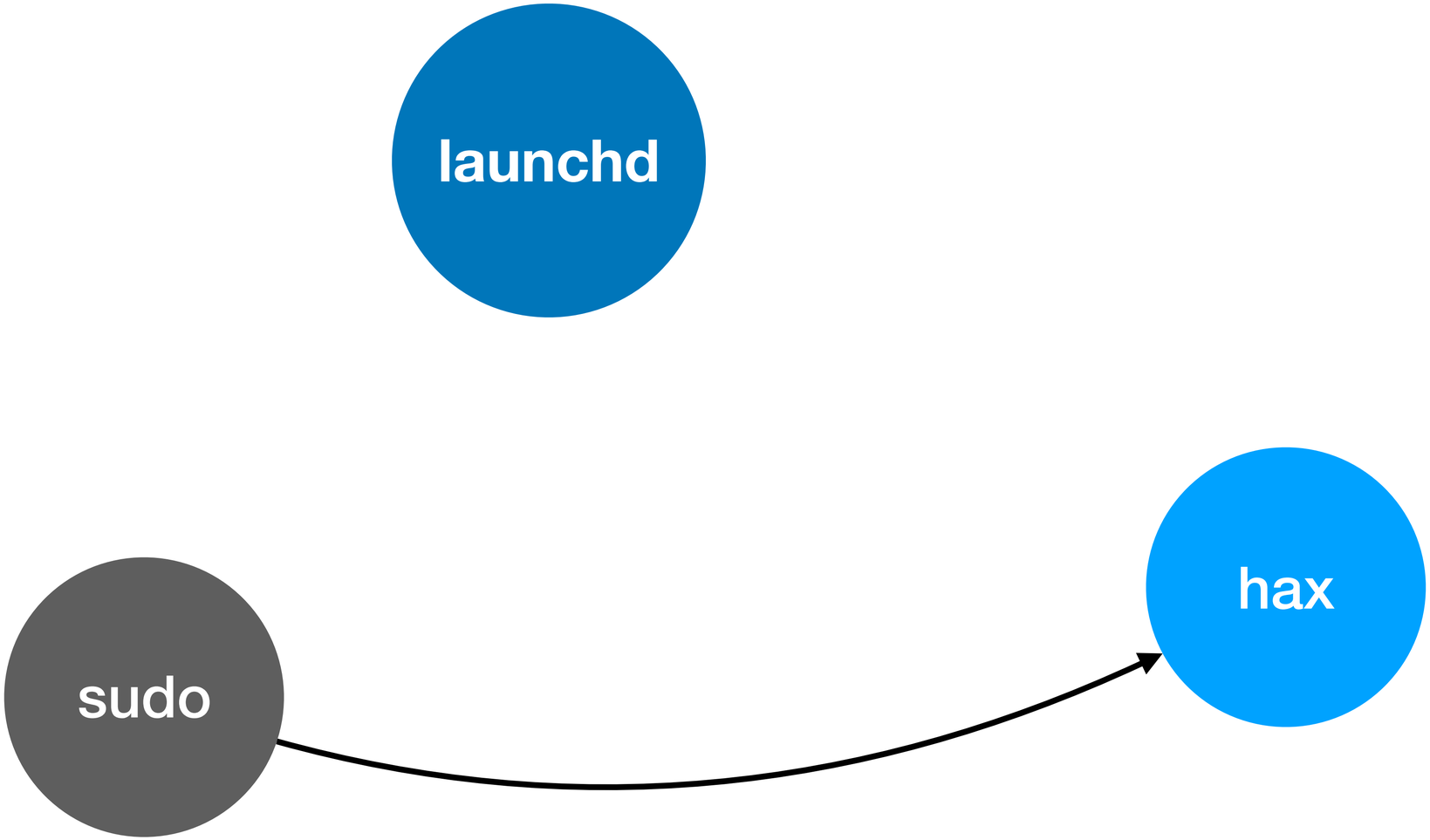
Passwordless Sudo



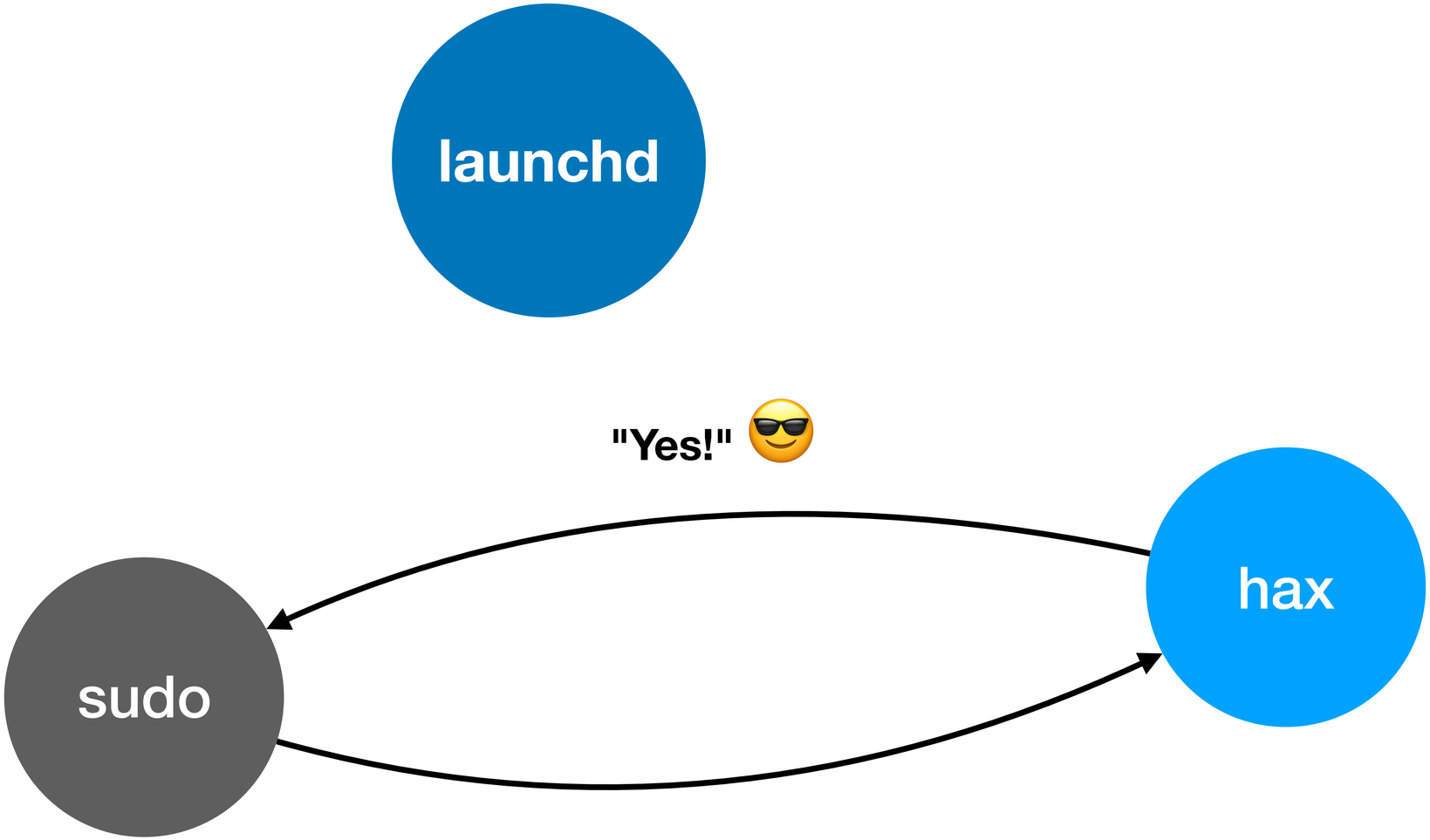
Passwordless Sudo



Passwordless Sudo

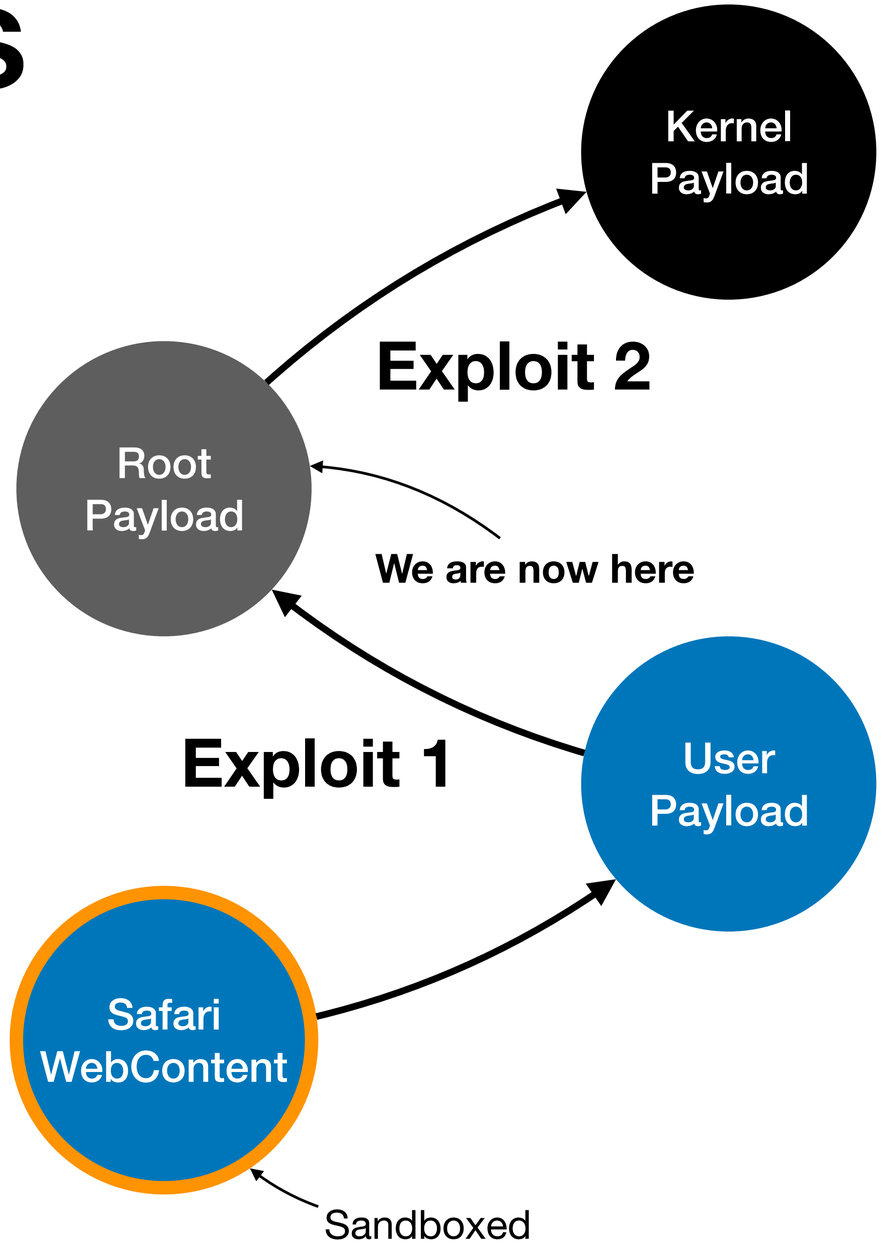


Passwordless Sudo



Status

- Have root privileges now \o/
- Goal: get into kernel
- On macOS: root -> kernel is a privilege boundary since introduction of SIP
- Loading kernel modules requires `com.apple.rootless.kext-management` entitlement
  - Possessed e.g. by /usr/bin/kextutil*



* See http://newosxbook.com/ent.jl?ent=com.apple.rootless.kext-management&osVer=MacOS13

`kextutil`

- Tool used to load kernel extensions ("kext") into the kernel
- Kext will only be loaded if:
  - `kextutil` is running as root ✔
  - The kext has a valid signature
  - The signature chain is rooted in an apple certificate
  - The kext has been approved by the user (https://developer.apple.com/library/archive/technotes/tn2459/_index.html)

Signature Verification

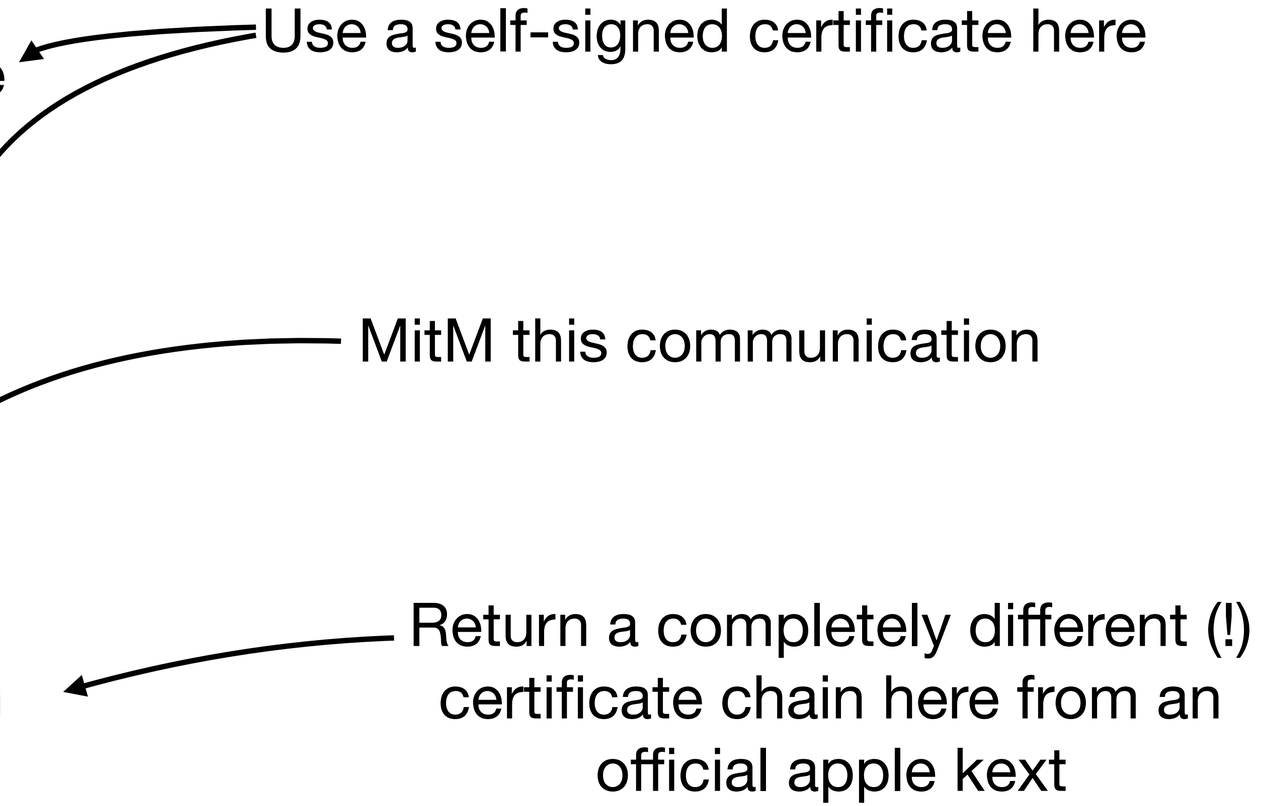
`kextutil` verification steps:

1. Extract the certificate from the provided kext bundle
2. Verify that the kext is signed with the attached certificate
3. Ask `trustd` to retrieve and validate the certificate chain from the supplied certificate
4. Verify that the certificate chain returned from `trustd` is anchored in an apple certificate

Signature Verification

`kextutil` verification steps:

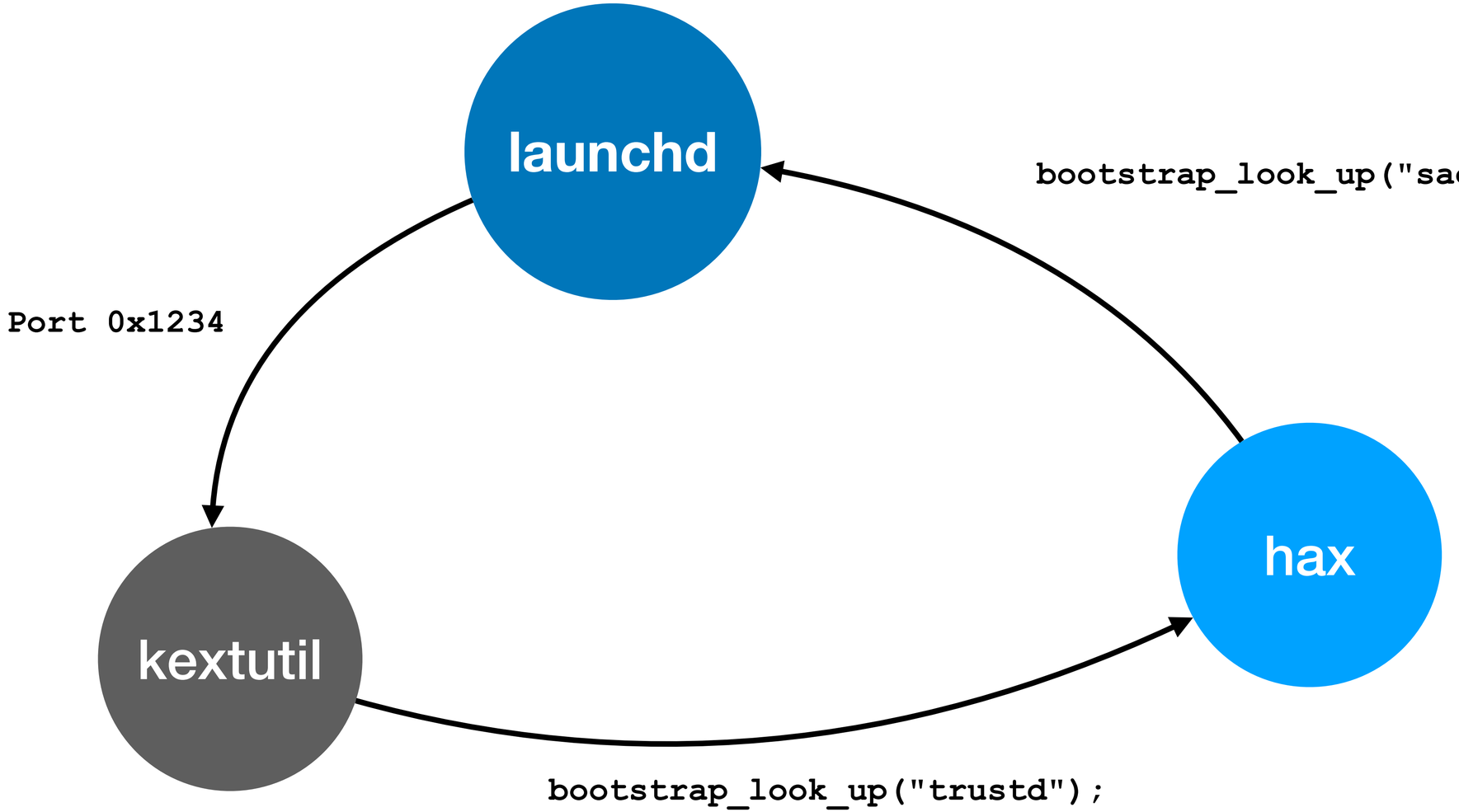
1. Extract the certificate from the provided kext bundle
2. Verify that the kext is signed with the attached certificate
3. Ask `trustd` to retrieve and validate the certificate chain from the supplied certificate
4. Verify that the certificate chain returned from `trustd` is anchored in an apple certificate



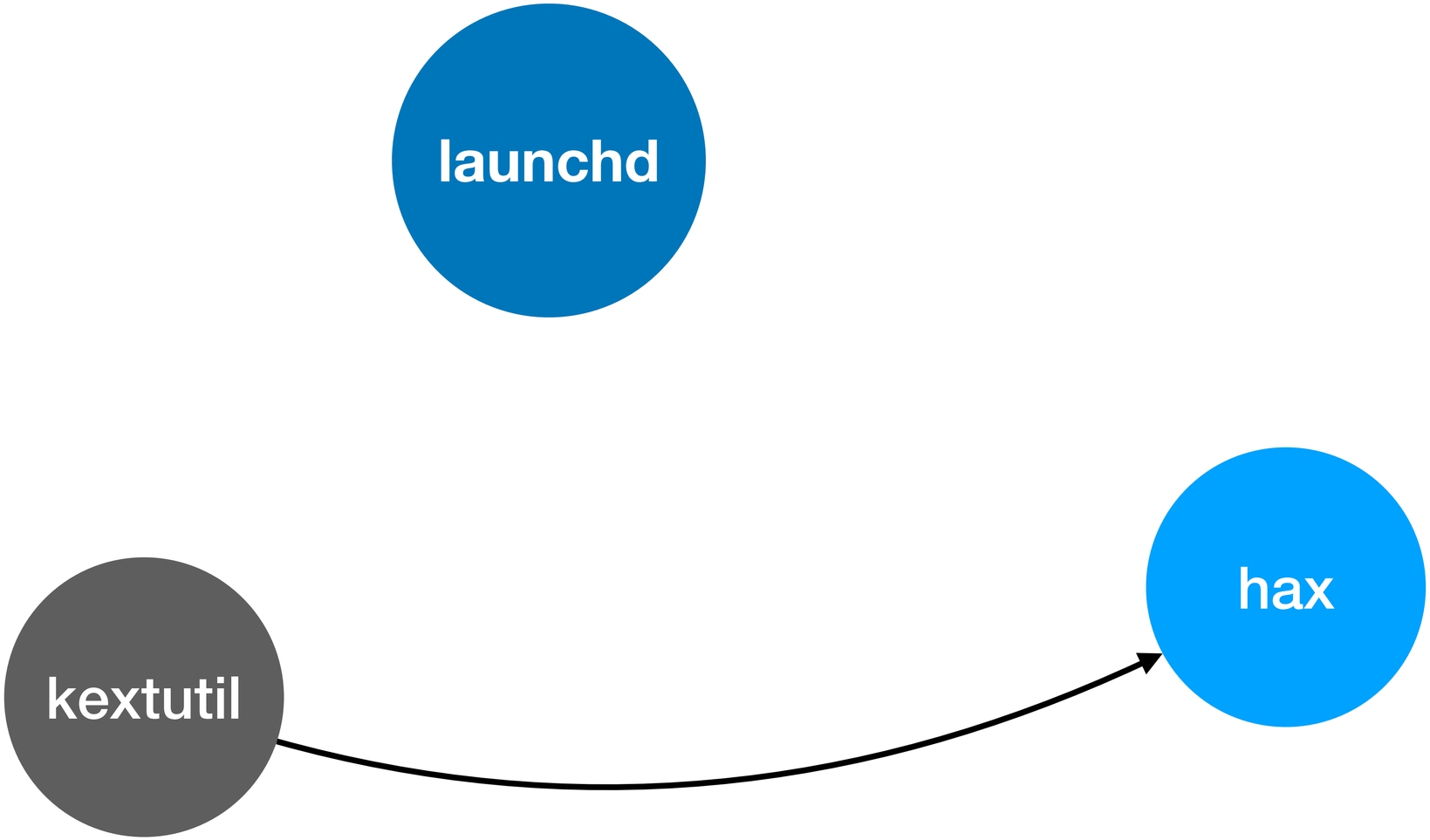
Tricking `kextutil`

... same setup as before

Tricking `kextutil`

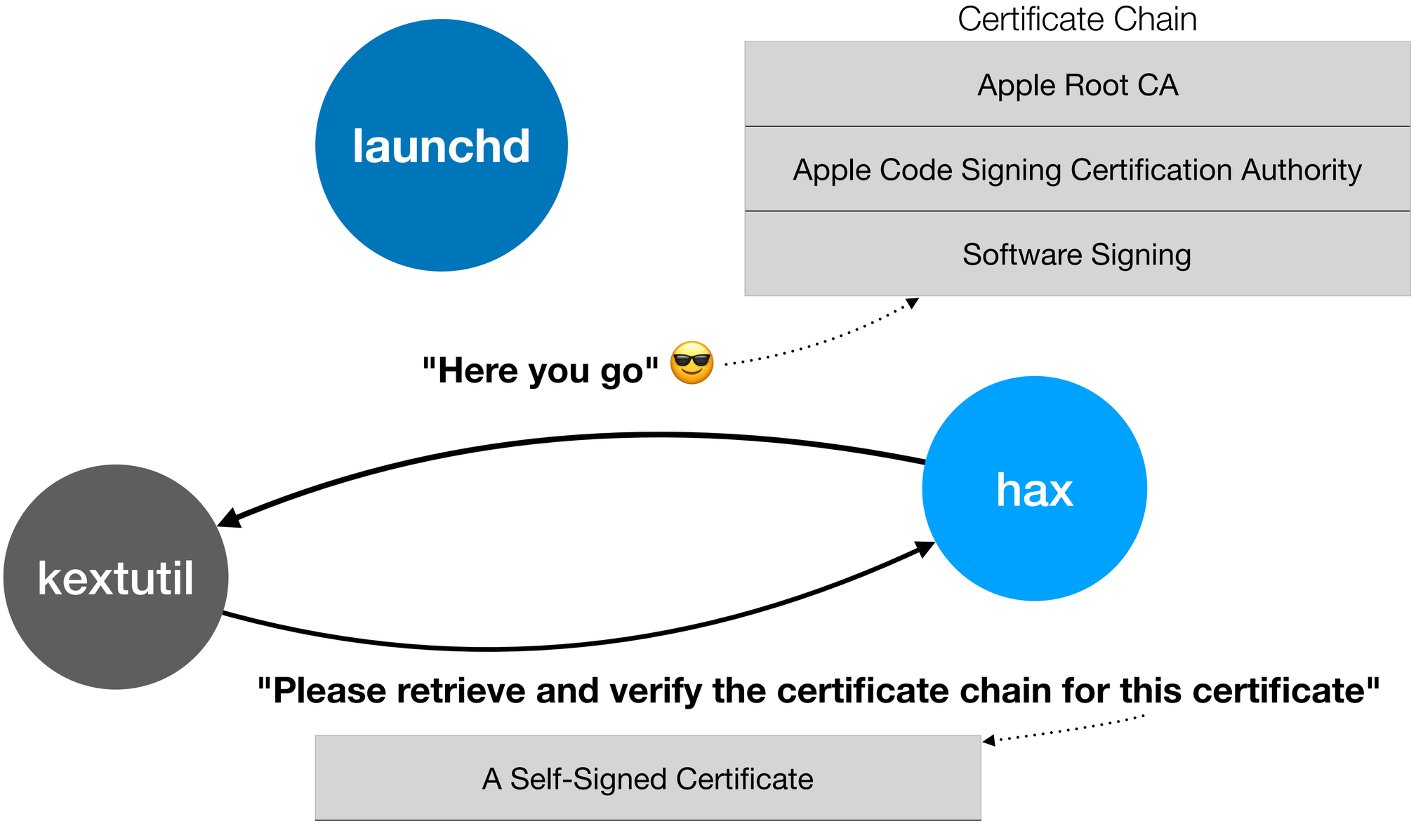


Tricking `kextutil`



"Please retrieve and verify the certificate chain for this certificate here"

Tricking `kextutil`

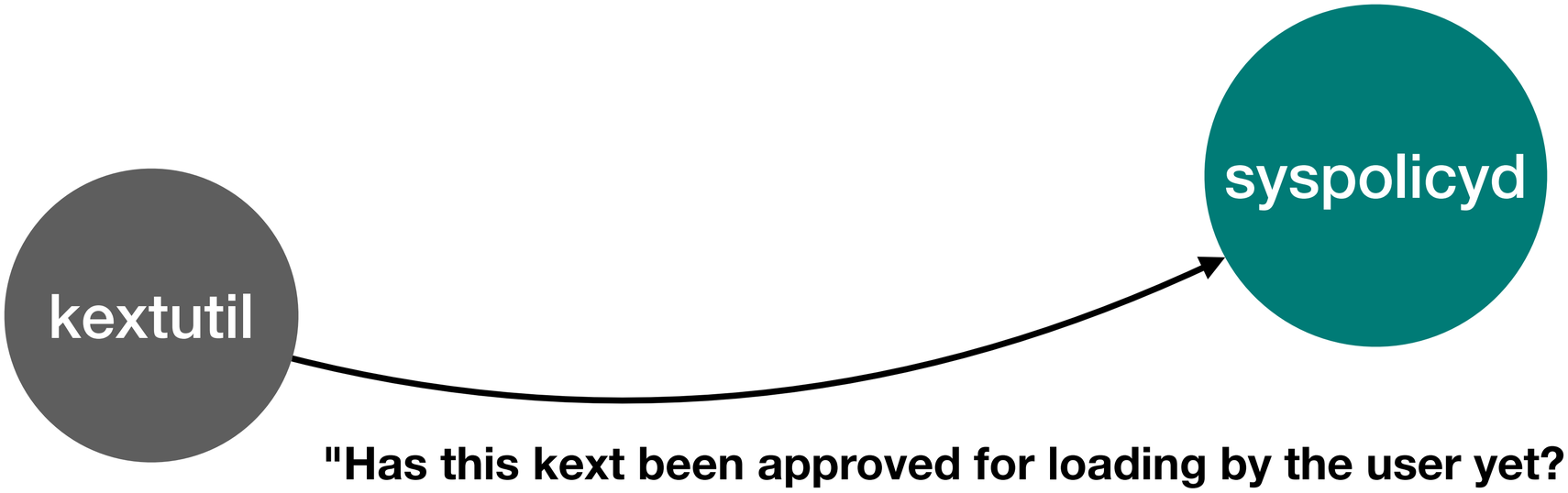


`kextutil`

- Tool used to load kernel extensions ("kext") into the kernel
- Kext will only be loaded if:
  - `kextutil` is running as root ✓
  - The kext has a valid signature ✓
  - The signature chain is rooted in an apple certificate ✓
  - The kext has been approved by the user (https://developer.apple.com/library/archive/technotes/tn2459/_index.html)

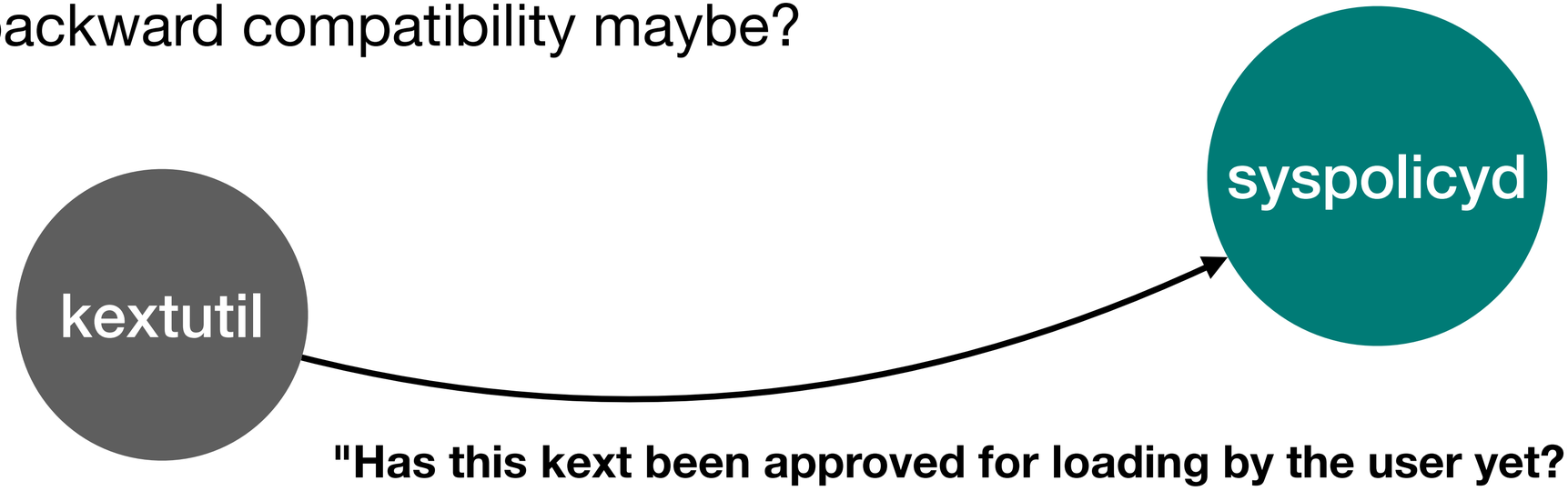
User-Approved Kext Loading

"macOS High Sierra 10.13 introduces a new feature that requires user approval before loading newly-installed third-party kernel extensions (KEXTs). When a request is made to load a KEXT that the user has not yet approved, the load request is denied."

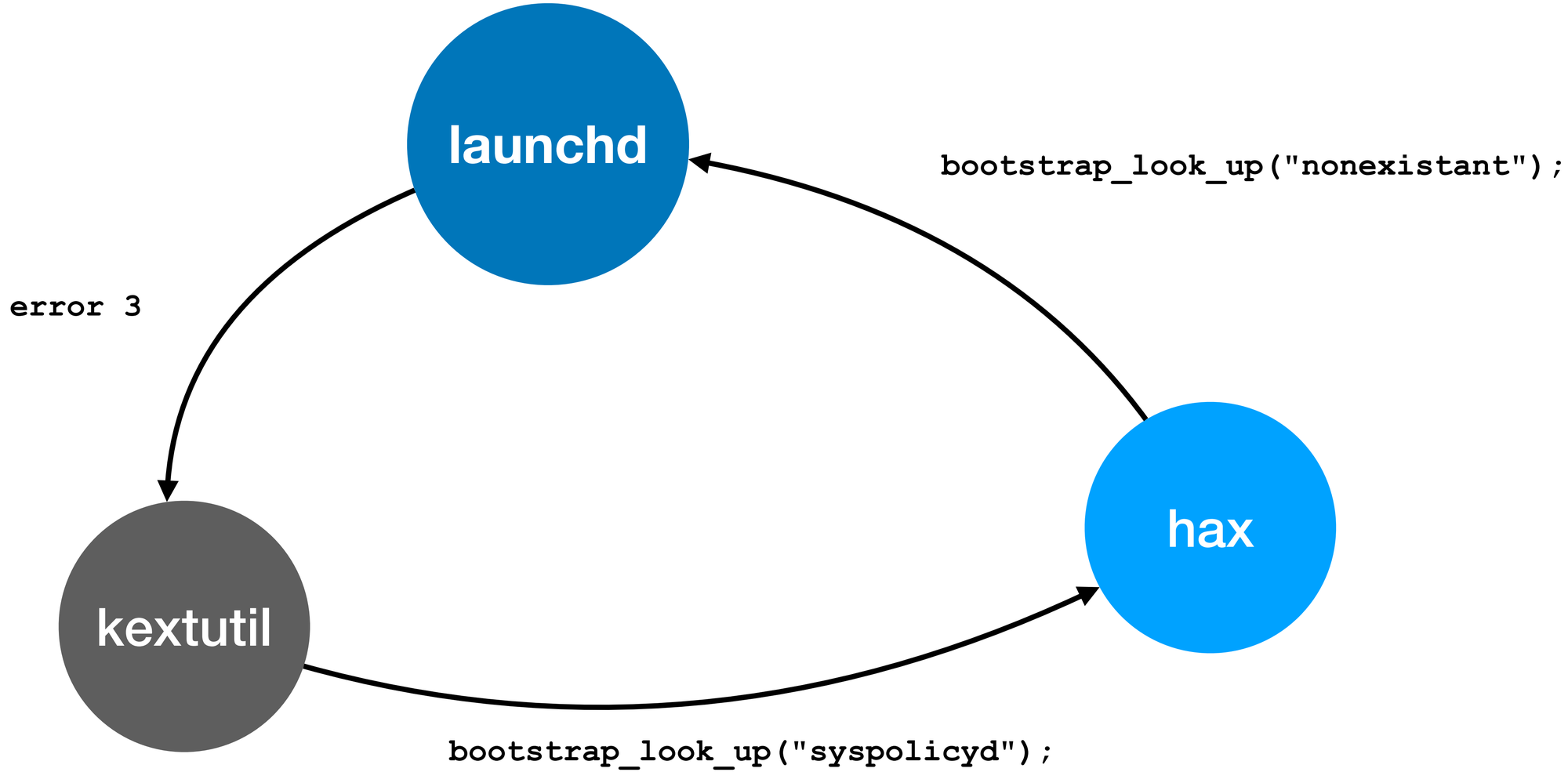


User-Approved Kext Loading

- Either spoof reply from syspolicyd
- Or prevent mach lookup of syspolicyd, in which case `kextutil` will also load the kext
  - For backward compatibility maybe?



Tricking `kextutil`



Demo

https://youtu.be/63MKVqdEJ6k

libspc

- Hacky reimplementation of XPC protocol
- Quite flexible, supports most relevant features
- Used to e.g. implement XPC intercepting and bridging for the exploits

```
while (1) {
    spc_message_t* msg = spc_recv(bridge->receive_port);

    msg->local_port.name = MACH_PORT_NULL;
    msg->remote_port.name = bridge->send_port;
    // Hack: replace "error: 5000" with "error: 0" to indicate success
    spc_dictionary_item_t* item = spc_dictionary_lookup(msg->content, "error");
    if (item)
        item->value.value.u64 = 0;

    spc_send(msg);
    spc_message_destroy(msg);
}
```

Summary

- OS's have gotten more complex
- Fun logic bugs out there
- Powerful exploitation possible with IPC bugs
- Full Pwn2Own exploit chain @ https://github.com/saelo/pwn2own2018

References

- libxpc.dylib and https://opensource.apple.com/source/xnu/
- https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html
- https://thecyberwire.com/events/docs/IanBeer_JSS_Slides.pdf
- https://github.com/bazad/blanket
- https://robert.sesek.com/2014/1/changes_to_xnu_mach_ipc.html