secmodel_sandbox
An application sandbox for NetBSD

Stephen Herwig

University of Maryland
Sandboxing: limiting the privileges of a process

Two motivations

- Running untrusted code in a restricted environment

- Dropping privileges in trusted code so as to reduce the attack surface in the event of an unforeseen vulnerability
many os-level implementations

- systrace
- SELinux
- AppArmor
- seccomp(-bpf)
- Apple’s Sandbox (formerly Seatbelt)
- Capsicum
- OpenBSD’s pledge syscall

Rich design space:

- which use cases are supported?
- footprint (system-wide or process-wide)
- are policies embedded in program or external?
- when are policies loaded?
- expressiveness of policies?
- portability
secmodel_sandbox high-level design

- Implemented as a kernel module
- Sandbox policies are Lua scripts
- A process sets the policy script via an ioctl
- The kernel evaluates the script using NetBSD’s experimental in-kernel Lua interpreter
- The output of the evaluation are rules that are attached to the process’s credential and checked during privileged authorization requests
- Sandboxes are inherited during fork and preserved over exec
- Processes may apply multiple policies: the sandbox is the union of all policies
- Policies can only further restrict privileges
- Rules may be boolean or Lua functions (functional rules)
- Functional rules may be stateful and may dynamically create new rules or modify existing rules
secmodel_sandbox properties

- Sandboxes are inherited during fork and preserved over exec

- Processes may apply multiple policies: the sandbox is the union of all policies

- Policies can only further restrict privileges

- Rules may be boolean or Lua functions (functional rules)

- Functional rules may be stateful and may dynamically create new rules or modify existing rules
sandbox policies: blacklist

**Policy**

```c
sandbox.default('allow');

-- no forking
sandbox.deny('system.fork')

-- no networking
sandbox.deny('network')

-- no writing to files
sandbox.deny('vnode.write_data')
sandbox.deny('vnode.append_data')

-- no changing file metadata
sandbox.deny('vnode.write_times')
sandbox.deny('vnode.change_ownership')
sandbox.deny('vnode.write_security')
```

**Program**

```c
main()
{
  /* initialize */
  ...

  sandbox(POLICY);

  /* process loop */
  ...

  return (0);
}
```
sandbox policies: functional rules

sandbox.default('deny')  
-- allow reading files
sandbox.allow('vnode.read_data')  
-- only allow writes in /tmp
sandbox.on('vnode.write_data',
    function(req, cred, f)
        if string.find(f.name, '/tmp/') == 1 then
            return true
        else
            return false
        end
    end)

-- only allow unix domain sockets
sandbox.on('network.socket.open',
    function(req, cred, domain, typ, proto)
        if domain == sandbox.AF_UNIX then
            return true
        else
            return false
        end
    end)
sandbox-exec

```c
int main(int argc, char *argv[]) {
    sandbox_from_file(argv[0]);
    execv(argv[1], &argv[1]);
    return (0);
}
```

$ sandbox-exec no-network.lua /usr/pkg/bin/bash
$ wget http://www.cs.umd.edu/
wget: unable to resolve host address `www.cs.umd.edu'
kauth

- kernel subsystem that handles all authorization requests within the kernel

- clean room implementation of subsystem in macOS

- separates security policy from mechanism
kauth requests

request := (scope, action [, , subaction])
kauth requests

request := (scope, action [, subaction])

Example:
creating a socket => (network, socket, open)
kauth request to syscall mapping

Some kauth requests map directly to a syscall:

```
system.mknod  =>  mknod
```

Some kauth requests map to multiple syscalls:

```
process.setsid  =>  {setgroups setlogin setuid
                     setuid setreuid setgid setegid setregid}
```

Some syscalls trigger one of several kauth requests, depending on the syscall arguments:

```
mount(MNT_GETARGS)  =>  system.mount.get
mount(MNT_UPDATE)   =>  system.mount.update
```

Many syscalls do not trigger a kauth request at all:

```
accept  close  dup  execve  flock  getdents  getlogin
getpeername  getpid  getrlimit  getsockname  .  .  .
```
kauth request flow

kauth uses an observer pattern.

syscall(arg1, ..., argn)

user space

kernel space

**kauth listener #1**

```c
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
    ...
    return (KAUTH_RESULT_ALLOW);
}
```

**kauth listener #2**

```c
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
    ...
    return (KAUTH_RESULT_ALLOW);
}
```

**syscall handler**

```c
kauth_authorize_action(cred, req, ctx);
```

**kauth**

```c
foreach (listener in scope) {
    error = listener->cb(cred, op, ctx);
    if (error == KAUTH_RESULT_ALLOW)
        allow = 1;
    else if (error == KAUTH_RESULT_DENY)
        fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

list of network scope listeners

[ lists for other scope listeners ]
Subsystems interested in kauth requests register with kauth via `kauth_listen_scope()`.

```
syscall(arg1, ..., argn)
```

**kauth**

```
kauth_authorize_action(cred, req, ctx);
```

```
foreach (listener in scope) {
    error = listener->cb(cred, op, ctx);
    if (error == KAUTH_RESULT_ALLOW)
        allow = 1;
    else if (error == KAUTH_RESULT_DENY)
        fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

```
list of network scope listeners
[ lists for other scope listeners ]
```
kauth request flow

Most syscalls issue an authorization request in their corresponding handler via `kauth_authorize_action()`.

```c
syscall(arg1, ..., argn)
```

```c
foreach (listener in scope) {
    error = listener->cb(cred, op, ctx);
    if (error == KAUTH_RESULT_ALLOW)
        allow = 1;
    else if (error == KAUTH_RESULT_DENY)
        fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

**kauth request flow**

_kauth_authorize_action()_ iterates through each listener for the given scope, calling that listener’s callback.

```
syscall(arg1, ..., argn)
```

### User space

#### kauth listener #1

```c
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);
int cb(cred, op, ctx) {
  ... return (KAUTH_RESULT_ALLOW);
}
```

#### kauth listener #2

```c
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);
int cb(cred, op, ctx) {
  ... return (KAUTH_RESULT_ALLOW);
}
```

### Kernel space

#### syscall handler

```c
foreach (listener in scope) {
  error = listener->cb(cred, op, ctx);
  if (error == KAUTH_RESULT_ALLOW)
    allow = 1;
  else if (error == KAUTH_RESULT_DENY)
    fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

#### kauth

```c
kauth_authorize_action(cred, req, ctx);
```

_list of network scope listeners_

[ lists for other scope listeners ]
kauth request flow

Generally, if any listener returns DENY, the request is denied; if any returns ALLOW and none returns DENY, the request is allowed; otherwise, the request is denied.

user space

syscall(arg1, ..., argn)

kernel space

kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);
```  
```
int cb(cred, op, ctx) {
    . . .
    return (KAUTH_RESULT_ALLOW);
}
```  

kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);
```  
```
int cb(cred, op, ctx) {
    . . .
    return (KAUTH_RESULT_ALLOW);
}
```  

syscall handler

```
kauth_authorize_action(cred, req, ctx);
```  
```
foreach (listener in scope) {
    error = listener->cb(cred, op, ctx);
    if (error == KAUTH_RESULT_ALLOW)
        allow = 1;
    else if (error == KAUTH_RESULT_DENY)
        fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```  

kauth

Generally, if any listener returns DENY, the request is denied; if any returns ALLOW and none returns DENY, the request is allowed; otherwise, the request is denied.

list of network scope listeners

[ lists for other scope listeners ]
secmodel

A security model (secmodel) is a small framework for managing a set of related kauth listeners. Fundamentally, it presents a template pattern:

```
static kauth_listener_t l_system, l_network, ...;

void secmodel_foo_start(void)
{
    l_system = kauth_listen_scope(KAUTH_SCOPE_SYSTEM, secmodel_foo_system_cb, NULL);
    l_network = kauth_listen_scope(KAUTH_SCOPE_NETWORK, secmodel_foo_network_cb, NULL);
    ...;
}

void secmodel_foo_stop(void)
{
    kauth_unlisten_scope(l_system);
    kauth_unlisten_scope(l_network);
    ...;
}
```
The sandbox module registers listeners for all kauth scopes.
Applications link to libsandbox. Calls to sandbox() issue an ioctl to `/dev/sandbox`, specifying the policy script.
The sandbox module services the ioctl, creates a sandbox, initializes the sandbox with the script’s policy rules, and attaches the sandbox to the process’s cred.

**Diagram:**
- Application
  - Libsandbox
    - Sandbox (script)
  - /dev/sandbox
- Proc
- Cred
  - Uid
  - Groups
  - Specificdata
- Secmodel_sandbox module
  - Sandbox
    - Policy 1
Subsequent calls to `sandbox()` add new policies. The sandbox is collectively the union of all of its policies.
secmodel_sandbox design

When a syscall emits a kauth request, the secmodel_sandbox’s listener checks if the process’s cred has a sandbox; if so, it evaluates the request against all policies.
bsd44 is the default security model, and is composed of three separate models: suser, securelevel, and extensions.
suser implements the traditional root user as the user with effective-id 0.

Each listener is a whitelist: if the requesting cred is root, then the listeners return KAUTH_RESULT_ALLOW; otherwise, KAUTH_RESULT_DEFER.
**stock secmodels**

**securelevel** is a system-global policy that restricts certain operations for all users, including root.

Each listener is a **blacklist**: request decisions default to **KAUTH_RESULT_DEFER** unless explicitly forbidden, in which case the model returns **KAUTH_RESULT_DENY**.
extensions grant additional privileges to ordinary users, such as user-mounts and user control of CPU sets, or enable isolation measures, such as curtain mode.

extensions is implemented as a mix of blacklists and whitelists.

```
 stock secmodels
```

```
bsd44  
|-- suser  
|-- securelevel  
    `-- extensions
```
defer revisited

While all listeners returning DEFER usually results in a DENIED request, for the vnode scope, the last resort decision is based on traditional BSD 4.4 file access permissions.

In order to not allow elevation of privileges, secmodel_sandbox converts sandbox policy decisions of ALLOW to DEFER.

-- if not internally converted to DEFER, would allow
-- reading any file
sandbox.allow('vnode.read_data')

-- if not internally converted to DEFER, would allow
-- user to load and unload modules
sandbox.allow('system.module')
sandbox implementation

In the kernel, a policy has two main items:
- a Lua state (Lua virtual machine)
- ruleset
Before secmodel_sandbox evaluates the Lua script in the lua_State, secmodel_sandbox populates the lua_State with the sandbox functions and constants.
sandbox implementation

Each sandbox Lua function is a closure that contains a pointer back to the policy. In Lua terminology, the policy is a light userdata upvalue.

lua_State

sandbox_lua_default()
sandbox_lua_deny()
sandbox_lua_allow()
sandbox_lua_on()

policy

ruleset
sandbox implementation

When a `sandbox.default()`, `sandbox.allow()`, or `sandbox.deny()` function is evaluated in a script, the corresponding C function accesses the ruleset from the policy upvalue, and stores the decision for that rule.

```
sandbox.default('allow')
```
sandbox implementation

When a `sandbox.default()`, `sandbox.allow()`, or `sandbox.deny()` function is evaluated in a script, the corresponding C function accesses the ruleset from the policy upvalue, and stores the decision for that rule.

```lua
sandbox.deny('system.mount')
```

```
lua_State
sandbox_lua_default()
sandbox_lua_deny()
sandbox_lua_allow()
sandbox_lua_on()
```

```
policy
```

```
ruleset
default
allow
```

```
system
mount
deny
```
sandbox implementation

When a `sandbox.on()` rule is evaluated, the corresponding C function stores the Lua callback function for the rule in the Lua Registry.

```
sandbox.on('network.socket.open', function() ... end)
```
sandbox implementation

During a kauth request, secmodel_sandbox looks in the ruleset for the best matching rule.

request:  (system, time, adjtime)
matches:  default rule
decision:  allow

lua_State
sandbox_lua_default()
sandbox_lua_deny()
sandbox_lua_allow()
sandbox_lua_on()

Registry[]
sandbox implementation

During a kauth request, secmodel_sandbox looks in the ruleset for the best matching rule.

**request:** (system, mount, update)
**matches:** system.mount rule
**decision:** deny

---

lua_State

sandbox_lua_default()
sandbox_luadeny()
sandbox_lua_allow()
sandbox_lua_on()

Registry[]

---

policy

ruleset
default

allow

network

socket

open

idx

system

mount
deny
multiple policies

A process’s sandbox may have multiple policies.

policies are isolated; each has its own lua_State and ruleset.

During a kauth request for a process, each policy is evaluated. In effect, a sandbox is a per-process kauth listener.
multiple sandboxes

Policy_1

sandbox.
default(‘deny’)
allow(‘vnode.read’)  # needed for sandbox() ioctl
allow(
  ‘device.rawio_spec.rw’
)

Policy_2

sandbox.default(‘deny’)
process forking

A process contains a pointer to a credential.
Normally, when the parent forks, the child process points to the same credential, and the credential’s reference count is incremented.
Normal forking behavior has the unfortunate consequence that if the child creates a sandbox, the sandbox is also applied to the parent.
Moreover, if the parent then adds a policy, the policy is also applied to the child.
CASE 1: parent is not sandboxed and child creates a sandbox

secmodel_sandbox creates a new cred for the child when the child creates a sandbox.
process forking

Each process is then free to create its own sandboxes.
CASE 2: parent already has a sandbox and forks
Child gets a new cred and a new sandbox. The child’s sandbox points to the parent’s newest policy. Policies are ref counted.
process forking

Each process is free to further add its own policies.
process forking

Modification of the forking behavior uses kauth’s cred scope, which notifies of events in a cred’s lifecycle.

fork emits a **KAUTH_CRED_FORK** event. secmodel_sandbox handles this event by duplicating the parent’s cred if the cred contains a sandbox.

Duplicating a cred emits a **KAUTH_CRED_DUP** event that secmodel_sandbox uses to create the sandbox in the child. The sandbox’s first member points to the most recent policy in the parent’s cred.
stateful policies

**Policy**

```markdown
local _ = sandbox
local nsocks = 0
_.default('allow')
_.on('network.socket.open',
  function()
    nsocks = nsocks + 1
    if nsocks > 1 then
      return false
    else
      return true
    end
  end)
end)
```

**Program**

```markdown
main()
{
  sandbox(POLICY);
  socket();

  /* any additional 
   * calls to socket() 
   * will fail 
   */
}
```
**dynamic policies**

### Policy

```javascript
local _ = sandbox;
_.default('allow')
_.deny('vnode')
.on('process.signal',
  function(req, cred, p, sig)
    if sig == _.SIGUSR1 then
      _.allow('vnode')
      _.deny('network')
    end
  return true
end)
```

### Program

```javascript
main()
{
  signal(SIGUSR1, noop);
  sandbox(POLICY);
  /* network, but not fs */
  data = wget();
  kill(getpid(),SIGUSR1);
  /* fs, but not network */
  read_file()
}
```

---

micro benchmarks

sandbox(POLICY)
for (i = 0; i < 10,000,000; i++) {
    syscall()
}

<table>
<thead>
<tr>
<th></th>
<th>setpriority</th>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>no sandbox</td>
<td>1.597</td>
<td>14.725</td>
</tr>
<tr>
<td>sandbox.allow()</td>
<td>2.281</td>
<td>17.439</td>
</tr>
<tr>
<td>sandbox.on()</td>
<td>46.356</td>
<td>51.644</td>
</tr>
</tbody>
</table>
OpenBSD’s pledge

int pledge(const char *promises, const char *paths[])

- POSIX syscalls grouped into categories
- restricts the process to the subset of POSIX as specified by the categories in promises
- If the process invokes a syscall outside of the promised subset, the process is killed
## OpenBSD’s pledge

<table>
<thead>
<tr>
<th>SyCalls</th>
<th>chown</th>
<th>cpath</th>
<th>dns</th>
<th>fattr</th>
<th>flock</th>
<th>inet</th>
<th>proc</th>
<th>tmppath</th>
<th>stdio</th>
<th>unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>chown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcntl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>listen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>socket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenBSD’s pledge

**cpath** allows syscalls taking a path argument that create or destroy the file at that path.

<table>
<thead>
<tr>
<th>Syscalls</th>
<th>chown</th>
<th>cpath</th>
<th>dns</th>
<th>fattr</th>
<th>flock</th>
<th>inet</th>
<th>proc</th>
<th>tmppath</th>
<th>stdio</th>
<th>unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>chown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcntl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>listen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>socket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenBSD’s pledge

socket may be allowed if one of dns, inet, or unix is pledged.

<table>
<thead>
<tr>
<th>Sycalls</th>
<th>chown</th>
<th>cpath</th>
<th>dns</th>
<th>fattl</th>
<th>flock</th>
<th>inet</th>
<th>proc</th>
<th>tmppath</th>
<th>stdio</th>
<th>unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>chown</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>chmod</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>flock</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fcntl</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fork</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>listen</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mkdir</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>socket</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>unlink</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
OpenBSD’s pledge

When a syscall is trapped, the kernel checks:

Has the process called `pledge`?
  
  **YES.** Has the process pledged any of the promises assigned to the syscall?
  
  **YES.** Invoke the specific syscall handler.
  
  **NO.** Kill the process.

Richer syscalls require additional argument/context checking. Examples:

- `fcntl (stdio)`
  
  needs the `flock` promise if used for file locking.

- `unlink (cpath or tmppath)`
  
  If the file being deleted is outside of `/tmp`, then `cpath` is required.

- `socket (dns, inet, or unix)`
  
  The socket’s domain must match a promise.
emulating pledge with secmodel_sandbox

Ongoing effort. Several challenges:

- kauth does not emit requests for many syscalls
  - memory-related functions, setsockopt, etc.

- slight but important platform differences
  - sendsyslog
  - SOCK_DNS

- semantic differences
  - secmodel_sandbox preserves sandbox across an exec, whereas pledge does not
summary

secmodel_sandbox is a new security model for NetBSD that allows per-process restriction of privileges.

Source code is available at:
www.cs.umd.edu/~smherwig/