secmodel_sandbox : An application sandbox for NetBSD (draft)

Stephen Herwig
University of Maryland, College Park

Abtract

We introduce a new security model for NetBSD – secmodel_sandbox – that allows per-process policies for restricting privileges. Privileges correspond to kauth authorization requests, such as a request to create a socket or read a file, and policies specify the sandbox's decision: deny, defer, or allow. Processes may apply multiple sandbox policies to themselves, in which case the policies stack, and child processes inherit their parent's sandbox. Sandbox policies are expressed in Lua, and the evaluation of policies uses NetBSD 7's experimental in-kernel Lua interpreter. As such, policies may express static authorization decisions, or may register Lua functions that secmodel_sandbox invokes for a decision.

1 Introduction

A process sandbox is a mechanism for limiting the privileges of a process, as in restricting the operations the process may perform, the resources it may use, or its view of of the system. Sandboxes address the dual problems of limiting the potential damage caused by running an untrusted binary, and mitigating the effects of exploitation of a trusted binary. In either case, the goal is to restrict a process to only the necessary privileges for the purported task, and, in the latter case, to also drop privileges when they are no longer needed.

Although NetBSD currently lacks a sandbox mechanism, sandbox implementations exist for various operating systems. systrace [5], a multi-platform mechanism used in earlier versions of NetBSD, and seccomp [2], a Linux-specific implementation, exemplify the approach of specifying a per-process system call policy, and use system call interposition to enforce the policy filter. For systrace, the policy format is systrace-specific, whereas seccomp specifies the policy as a BPF program. OpenBSD's pledge system call [4] offers a simplified interface for dropping privileges: OpenBSD groups the POSIX interface into categories, and allows processes to whitelist or *pledge* their use of certain categories; an attempt to perform an operation from a non-pledged category kills the process.

We implement an application sandbox for NetBSD, secmodel_sandbox, that allows per-process restriction of privileges. secmodel_sandbox plugs into the kauth framework, and uses NetBSD's support for in-kernel Lua [7] to both specify and evaluate sandbox policies. We are developing several facilities with secmodel_sandbox, such as a secure chroot and a partial emulation of OpenBSD's pledge system call.

2 NetBSD Overview

2.1 kauth

NetBSD 4.0 introduced the kauth kernel subsystem [3] – a clean room implementation of Apple's kauth framework [6] for OS X – to handle authorization requests for privileged operations. Privileged operations are represented as triples of the form *(scope, action, optional subaction)*. The predefined scopes are `system`, `process`, `network`, `machdep`, `device`, and `vnode`, each forming a namespace that is further refined by the action and subaction components. For instance, the operation to create a socket is identified by the triple `(network, socket, open)`, and the operation to read a file by `(vnode, read_data)`.

Some authorizations, such as `(process, nice)`, are triggered by a single system call (setpriority); some, such as `(system, mount, update)`, are triggered when a system call (mount) is called with specific arguments (the `MNT_UPDATE` flag); and others, such as `(system, filehandle)` may be triggered by more than one system call (fhopen and fhstat). Many system calls do not trigger a kauth request.

kauth uses an observer pattern whereby listeners register for operation requests for a given scope; when a request occurs, each listener is called.

Each listener receives as arguments the operation triple, the credentials of the object (typically, the process) that triggered the authorization request, as well as additional context specific to the request.

Each listener returns a decision: either `allow`, `deny`, or `defer`. If any listener returns `deny`, the request is denied. If at least one listener returns `allow` and none returns `deny`, the request is allowed. If all listeners return `defer`, the decision is scope-dependent. For all scopes other than the vnode scope, the result is to deny the authorization. For the vnode scope, the authorization request contains a "fall-back" decision, which nearly always specifies a decision conforming to traditional BSD4.4 file access permissions.

2.2 secmodel

While the NetBSD kernel source contains many listeners (typically in accordance with kernel configuration options), the secmodel framework offers a lightweight convention for developing and managing a set of listeners that represents a larger security model. By default, NetBSD uses secmodel_bsd44, which implements the traditional security model based on 4.4BSD, and which itself is composed of three separate models: secmodel_suser, secmodel_securelevel, and secmodel_extensions.

An important, subtle point with the default security model is that many authorization requests are deferred, relying on kauth's default behavior when all listeners return `defer` to fully implement the policy.

3 Design

We developed secmodel_sandbox as a loadable kernel module with companion user-space library `libsandbox`. By convention, we install the device file for secmodel_sandbox at `/dev/sandbox`.

A process interacts with secmodel_sandbox via the `sandbox(const char *script, int flags)` function of `libsandbox`. The argument `script` is a Lua script that specifies the sandbox policy. The `flag` argument specifies the action to take when a process attempts a denied operation: a value of `0` means that the operation returns an appropriate errno as dictated by kauth (typically `EACCES` for kauth's vnode scope and `EPERM` for all other scopes); a value of `SANDBOX_ON_DENY_KILL` specifies the pledge behavior of killing the process. The `sandbox` function packages these arguments into a struct and, via an ioctl call, passes the struct to `/dev/sandbox`.

secmodel_sandbox evaluates the Lua script in a Lua environment that is pre-populated with a `sandbox` Lua module. The `sandbox` Lua module allows a script to set policy rules via the following interface:

```
sandbox.default(result)
sandbox.allow(req)
sandbox.deny(req)
sandbox.on(req, func)
```

The `sandbox.default` function specifies a result of either `'allow'`, `'deny'`, or `'defer'`. The result is the sandbox's decision for any kauth request for which the script does not specify a more specific rule.

The `sandbox.allow` and `sandbox.deny` specify allow and deny rules, respectively, for the kauth request given as `req`.

The sandbox Lua module uses strings of the form `'scope.action.subaction'` to represent the requests; hence, a request to open a socket corresponds to the string `'network.socket.open'`, and a request to read a file to `'vnode.read_data'`. A script may specify a complete request name, or a prefix. When the process triggers an authorization request, secmodel_sandbox will select the policy rule that has the longest prefix match with the given request. As an example, a sandbox policy script of:

```
sandbox.default('deny')
sandbox.allow('network')
```

would allow any request in the network scope, but would deny requests from all other scopes.

The `sandbox.on` Lua function registers a Lua function `func` to be called for the given kauth request. The signature for `func` is:

```
func(req, cred, arg0, arg1, arg2, arg3)
```

where `req` is the kauth request that generated the callback, cred is a Lua table that represents the credentials of the requesting object or process, and the remaining arguments are request-specific. All parameters for `func` exist only in the Lua environment; manipulating the values does not affect the underlying C objects that they represent.

For many requests, the values for `arg0` through `arg3` are `nil`, as the kauth request carries no additional context. For the requests that do contain context, we translate the context into appropriate Lua values. For example, for the request `'network.socket.open'`, the arguments are Lua integers representing the arguments to the `socket` system call that triggered the request. For clarity in script writing, we pre-populate the `sandbox` Lua module with symbols for common constants, such as `sandbox.AF_INET` and `sandbox.SOCK_STREAM`. For requests in the process scope, `arg0` is a Lua table that represents a subset of the fields of the `struct proc` that is the target of the request, such as the `pid`, `ppid`, `comm` (program name), and `nice` value. Callback functions for the vnode scope receive as `arg0` a Lua table that contains the pathname and file status information of the target vnode. Completely representing the context with Lua values is an ongoing effort.

4 Sandbox Implementation

Our design and implementation of secmodel_sandbox considered several important requirements and features. First, while expressing rules in Lua is elegant, having to call into Lua to find a matching rule for each request is not. Thus, we implemented secmodel_sandbox so that-evaluating the policy script "compiles" the rules into a prefix tree, mimicking the natural hierarchy provided by the (scope, action, subaction) format of requests. Thus, secmodel_sandbox can quickly find a matching rule, and only needs to call into Lua for *functional* rules – rules specified as Lua functions via `sandbox.on`.

Second, we wanted to allow sandboxes to be *dynamic*; that is, allow a functional rule to set other rules. For example, a script might create rules based on the requesting credential, as in the following, which installs a functional rule for the network scope so that different rules may be created for the root user and for ordinary users:

```
sandbox.on('network', function(rule, cred)
  if cred.euid == 0 then
      sandbox.allow('network.bind')
      ...
  else
      sandbox.deny('network.bind')
      ...
  end
end)
```

Third, we had to be mindful of the subtleties of the default security model, particularly its dependency on kauth's default decisions when all listeners return defer, so as not to allow sandboxes to elevate a process's privileges beyond the default model. In a similar vein, we needed to isolate multiple sandboxes on a single process so that the process is not able to install a new sandbox that loosens or undoes a rule of an existing sandbox.

Finally, in order to ensure that child processes inherit the sandboxes of their parent, but that, after process creation, parent and child may apply additional sandboxes independently of one another, we had to extend the normal forking behavior.

4.1 Sandbox creation

When a process sets a sandbox policy via `libsandbox`, the kernel creates a new sandbox, represented as a `struct sandbox`. A sandbox contains two main items: a Lua state and a ruleset. The Lua state is the Lua environment in which secmodel_sandbox evaluates all Lua code for that particluar sandbox. The ruleset is a prefix tree that secmodel_sandbox searches during a kauth request to find the sandbox's matching rule.

Before secmodel_sandbox evaluates the policy script in the newly created Lua state, secmodel_sandbox adds the sandbox Lua functions (e.g., `sandbox.allow`) and constants (e.g., `sandbox.AF_INET`) to the state. Each sandbox Lua function is a closure that contains a pointer to the `struct sandbox`. In Lua terminology, the `struct sandbox` is a light userdata upvalue.

When the script calls a sandbox Lua function, the function – which is implemented in C code – performs argument checking, retrieves the ruleset from the `struct sandbox` upvalue, and inserts the rule and the rule's value into the ruleset.

For `sandbox.allow`, `sandbox.deny`, and `sandbox.default`, the rule's value is a trilean: one of `KAUTH_RESULT_ALLOW`, `KAUTH_RESULT_DENY`, or `KAUTH_RESULT_DEFER`, as defined in `sys/kauth.h`. For `sandbox.on`, the value is an index into Lua's registry. The Lua registry is a global table that can only be accessed from C code. When a script invokes `sandbox.on`, secmodel_sandbox stores the callback function at an unused index in the Lua registry, and the ruleset stores this index as the rule's value.

After evaluating the policy script, secmodel_sandbox attaches the `struct sandox` to the current process's credentials. The data that secmodel_sandbox attaches to a credential is in fact a list of `struct sandbox`'s, to support allowing a process to apply multiple sandboxes during the course of its execution. If the list does not exist, secmodel_sandbox first creates it and inserts the new sandbox; otherwise, the new `struct sandbox` is added to the existing list.

Storing `struct sandbox` as an upvalue supports the creation of dynamic rules; that is, a `sandbox.on` callback function that creates rules for other requests as part of its evaluation. If the callback function creates new rules by calling any of the `sandbox` Lua module functions, then the C implementations of these functions can immediately find the corresponding ruleset for the given Lua state.

4.2 Evaluating Authorization Requests

secmodel_sandbox registers listeners for all kauth scopes. When one of the secmodel_sandbox listeners is called, the listener checks whether a list of `struct sandboxs` is attached to the requesting credential. If a list is not attached, the listener defers; if a list is attached, secmodel_sandbox searches the ruleset of each `struct` sandbox for a value, calling into Lua if the value represents a registry index for a callback function.

If any sandbox in the list returns deny, secmodel_sandox returns deny for the request; if at least one sandbox returns allow and none returns deny, secmodel_sandbox returns defer, not allow as one would presume. The reason for "converting" allow to defer is due to subtleties in the implementation of kauth(9) and of the default security models that implement the traditional BSD4.4 security policy. In particular, since a large part of the traditional security model is implemented by having all listeners defer, and thus relying on kauth's "fallback" behavior, secmodel_sandbox must also defer, so as not to allow the elevation of privileges.

4.3 Process forking

In NetBSD, a process's credentials are represented by the kauth_cred_t type. The kauth framework emits events corresponding to a credential's life-cycle via the cred scope. As with other kauth scopes, listeners may register callback functions.

When a process forks, the normal behavior is for the parent and child to share the same kauth_cred_t, and to simply increment the credential's reference count. During the fork process, however, the kauth framework emits a fork event, thereby allowing for other behavior. For the fork event, the listener callback functions receive the struct proc of the parent and child, as well as the shared credential.

secmodel_sandbox registers a callback for credential events. During a fork event, secmodel_sandbox checks whether the credential contains a list of sandboxes. If yes, then secmodel_sandbox creates a new credential for the child process that is identical to the parent's credential, with the exception that the child credential contains a new list head for the list of sandboxes. Althought the list head of the parent and child are different, they both point to the same initial struct sandbox. Thus, each sandboxed process has its own kauth_cred_t and its own sandbox list head, but the individual struct sandboxs are shared among the related processes, and hence reference counted.

The handling of sandboxes in this manner means that the child is restricted by the same sandboxes as its parent at the time of the child's creation, but that after the child's creation, parent and child may add additional sandbox policies that do not affect the other process.

4.4 Mapping vnodes to pathnames

The request context for the the vnode kauth scope contains the vnode that is the target of the operation. For a sandbox policy, however, it is much more natural to work with pathnames rather than vnodes.

secmodel_sandbox uses methods similar to those in sys/kern/vfs_getcwd.c to attempt to retrieve a pathname. The method is to search for the basename of the vnode in the namei cache via cache_revlookup, and then walk back to the root vnode via interspersing calls to VOP_LOOKUP (to retrieve a parent vnode), and VOP_READDIR (to find the path name component of the child vnode). While we would expect the initial vnode to be present in the cache, an obvious weakness of this method is the reliance on a cache hit, which cannot be gauranteed.

4.5 Safeguards

Evaluating user-provided Lua scripts in the kernel raises a few concerns. An obvious concern is denial-of-service caused by a Lua script with an infinite loop. While not yet implemented, the defense is straight-forward, and used in the Lua kernel module to handle creating Lua states with luactl.

In short, as part of its C API, Lua provides the function lua_sethook for an application to register a C hook function to be called at various Lua VM events. In particular, an application can register to receive a callback after every *n* Lua VM instructions. The approach is therefore to set a hook function to be called after some maximum number of VM instructions; if the hook is called, the hook stops execution of the Lua VM by calling lua_error. Lua allows only one hook function per Lua state; in order to "restore" the VM instruction count back to zero, the hook function must be reset before every evaluation of a Lua script or function.

Another concern is that the struct sandboxs or the callbacks registered via sandbox.on might be accessed and modified from Lua code. Values in the Lua registry and upvalues are, provided Lua's debug library is not loaded, only accessible from C code. secmodel_sandbox does not load the debug library. Moreover, secmodel_sandbox does not provide a require function or any other means to load additional Lua libraries.

5 Applications

In this section, we describe the tools and facilities we are developing with secmodel_sandbox.

5.1 Secure chroot

One application that we are developing is a secure chroot. In 2011, Aleksey Cheusov proposed the secmodel_securechroot security model [1]. secmodel_securechroot was developed as a kernel module, and once loaded, modifies the chroot system call to place additional restrictions on the chrooted process. The restrictions impose process containment by preventing process's with one root directory from viewing information about processes with a different root directory, as well as denying the chrooted process several system-wide operations, such as rebooting, modifying sysctls, or adding devices.

On NetBSD's `tech-kern` mailing list, there was disagreement over the exact operations that should be allowed and denied within a secure chroot. Moreover, some users expressed a desire to not override the default `chroot` behavior, but rather have an additional system call for secure chroot, so that users could choose the level of restriction for each chrooted process. While some of the changes to kauth needed to support `secmodel_sandbox` were merged into the NetBSD kernel source, the secmodel itself was not.

We are developing an implementation of secmodel_securechroot as an auxiliary function, `sandbox_securechroot`, in `libsandbox`, with an associated command-line tool. Development of the tool demonstrates that previously proposed secmodels can be implemented using secmodel_sandbox, and that secmodel_sandbox offers users flexibility in choosing the proper level of containment.

5.2 pledge

We are also developing the `libsandbox` auxiliary function `sandbox_pledge`, which attempts to emulate OpenBSD's pledge system call using secmodel_sandbox.

A sandbox policy that mimics pledge is essentially a whitelist: explicitly allowing actions that correspond to a given category, and denying all others. Certain categories are easily implemented. For instance, the pledge categories that correspond to the access and modification of file metadata, such as `rpath`, `wpath`, `fattr`, and `chown`, are, with small exceptions, handled by appropriate `vnode` scope rules. Similarly, categories that limit network access to certain domains, such as `inet` and `unix`, are covered by rules for `'network.bind'` and `'network.socket.open'`.

Several pledge categories, however, reference system calls that, in NetBSD, do not trigger kauth requests. For example, the `flock` category that allows file locking or the `dns` category that allows DNS network transactions, lack apprporiate kauth requests. As a result such categories cannot be implemented.

6 Conclusion

We have introduced and developed a new security model, secmodel_sandbox, for NetBSD that allows per-process specification and restriction of privileges. While several secmodels exist for NetBSD, secmodel_sandbox is novel in its use of NetBSD's in-kernel Lua interpreter to allow processes to express privileges, subject to the bounds of the traditional BSD4.4 security model. We designed secmodel_sandbox to limit excessive calls into Lua, to allow sandboxes to dynamically create rules during the execution of a process, to allow a process to specify multiple, isolated, sandboxes during the course of its execution, and to ensure that a child process inherits the sandbox of its parent. We are developing concrete, familiar, applications in order to demonstrate our design's ease and flexibility in developing secure software.

References

[1] Aleksey Cheusov. *RFC: New security model secmodel_securechroot(9)*. URL: `https://mail-index.netbsd.org/tech-kern/2011/07/09/msg010903.html`.

[2] Jake Edge. "A seccomp overview". In: (Sept. 2015). URL: `https://lwn.net/Articles/656307/`.

[3] Elad Efrat. "Recent Security Enhancements in NetBSD". In: *Proceeding of EuroBSDCon 2006*. 2006. URL: `http://www.netbsd.org/~elad/recent/recent06.pdf`.

[4] *pledge(2)*. OpenBSD 6.0.

[5] Niels Provos. "Improving Host Security with System Call Policies". In: *Proceeedings of the 12th Conference on USENIX Security Symposium - Volume 12*. 2003.

[6] *Technical Note TN2127: Kernel Authorization*. Tech. rep. Apple Inc., 2010. URL: `https://developer.apple.com/library/content/technotes/tn2127/_index.html`.

[7] Lourival Vieira Neto et al. "Scriptable Operating Systems with Lua". In: *Proceedings of the 10th ACM Symposium on Dynamic Languages*. 2014.