Gaols

Implementing Jails Under the kauth Framework

Christoph Badura
The NetBSD Foundation
bad@netbsd.org

Abstract

FreeBSD's jail facility provides a light-weight form of virtualization by restricting access to the file system, to privileged system calls, to network resources and the ability to see and interfere with processes in other jails for the imprisoned processes.

NetBSD 4.0 introduced a clean-room implementation of Apple's kauth(9) framework. This framework replaces the open-coded checks for appropriate privilege with standardized calls for authorization of specific enumerated actions. Thus allowing fine-grained control over which actions are allowed and which are denied. The kauth(9) framework also provides a mechanisms to dynamically add and remove security modules that take part in the authorization process and can alter the default behaviour.

This paper explores the implementation of FreeBSD style jails under the NetBSD kauth(9) framework. Jail-like functionality was chosen because it is both relatively simple to implement and at the same time stresses the limits of the framework because of the need to not only return "allowed"/"not allowed" decisions but also to modify network addresses in the callers environment.

This paper presents work in progress.

1. Introduction

FreeBSD Jails

FreeBSD 4.0 introduced a light-weight virtualization facility called jails [KAWA2000]. The facility builds on the chroot concept and extends it by imposing additional restrictions on the *imprisoned* processes:

- File system access:
  Jailed processes run chrooted and have access to the files below the chroot point only.
- Network access:
  Process in a jail are allowed to bind sockets to a single specified Ipv4 address only.
- Restricted root rights:
  Privileged system calls that would affect the host system or other jails are forbidden.
- Restricted visibility of resources outside prison:
  Jailed processes cannot see processes and sockets that are in different jails.

Jails don't provide full virtualization because they do not restrict usage of resources like CPU, memory, I/O and network bandwidth.

Jails predate the FreeBSD MAC kernel framework which provides fine-grained control over and authorization for privileged actions. Interestingly, the jail facility was not re-implemented as a MAC module, although it uses some of the services of the MAC framework, when that is compiled into they system. Also, the MAC system has special knowledge of jail system.

The kauth Framework

The kauth(9) framework was introduced by Apple in Mac OS X 10.4 Tiger [TN2127]. Elad Efrat provided a clean-room implementation of the framework for NetBSD 4.0 [EFRAT2006].

On the caller side this framework replaces the open-coded calls to suser(9) etc. that check that the requester has appropriate privileges with calls to a generic authorization framework asking permission for specific, named actions. On the kauth(9) framework side the authorization actions were, for the first time, explicitly enumerated and run through a modular, pluggable, and extensible system.

The system can be extended by linking additional security models into the kernel that participate in the authorization processes and registering them with the authorization framework. Security models can also be added at runtime as loadable kernel modules.

The proponents of the kauth(9) framework claim the following benefits:

- Reducing coding errors at the calling places by replacing the maze of a twisty little open-coded calls, all slightly different, with easy to understand “boilerplate” code to check authorization for the requested action.
- Reducing coding errors in the authorization modules by factoring out common code into small, modular, and easy to understand methods.
- Introducing a modular and extensible framework with pluggable security models. The claim is that this allows new security models to be implemented that refine and extend the traditional Unix security model.
- The ability to completely replace the default Unix security model in the operation system with an alternate model.

Role based access control (RBAC) and mandatory access control (MAC) are the usual examples cited for candidate security models. However, they are complex and difficult to implement. To the author's knowledge, no implementation has been attempted to date for the kauth(9) framework in NetBSD.

The first two benefits have been proven almost immediately by implementing kauth(9) and converting the kernel to the new framework. However, until to date nobody has tried to implement a new security model that extends the existing Unix security model and does something new or otherwise unforeseen by the designers of the kauth(9) interface.

One of the important factors in the success of Unix and the fact that it is still extremely competitive in the market 40 years after its invention is that its architecture and interfaces are so artfully designed that has been possible to almost seamlessly integrate new functionality into the system that the original designers had not anticipated. E.g. demand paged virtual memory, sockets and networking, virtual file systems, etc.

When a new subsystem is introduced into the kernel it is therefore important to exercise its capabilities and find out if it adds real new capabilities to the system that its designers haven't anticipated or if it is merely a factoring out of existing code that doesn't open any new possibilities.

This paper explores the implementation of a FreeBSD jail(2)-like security model under the kauth(9) framework. The jail API is a particularly good test case for stressing the design of the interface because it imposes only a few restrictions in addition to the traditional Unix security model. However some of these restrictions require non-trivial side-actions that were probably not foreseen by the designers of the kauth(9) interface. For example mapping the INADDR_ANY address to an IP address assigned to the jail for certain socket operations.

2. Background

Kauth Basics

With the kauth(9) framework the open-coded calls in the kernel that check for sufficient privilege are replaced with calls to the function *kauth_authorize_action()* to authorize the requests.[1] The arguments to the function are:

- a named scope
- the credentials of the entity asking for authorization
- a numeric constant identifying the action in the scope to be authorized
- up to 4 context specific arguments giving further details about the request

kauth_authorize_action() dispatches the request to the listeners that the security models interested in the particular scope have registered with the framework. The listeners examine the request and return one of KAUTH_RESULT_ALLOW, KAUTH_RESULT_DENY, or KAUTH_RESULT_DEFER indicating that they approve the request, deny it, or have no opinion. If at least one listener denies the request, *kauth_authorize_action()* will deny the authorization.

Typical scopes are the system scope, the process scope, the network scope, and the generic scope.

The kauth(9) framework provides a standard way for security models to attach model-specific data to credentials. There is also a special credentials scope for managing this security model specific data when credentials are copied or freed.

Differences to FreeBSD MAC framework

In the kauth(9) framework additional context-dependent information is usually passed to the authorization modules to allow making decisions based on this information.

Also, the kauth(9) framework provides a standard way to attach additional security model specific data to in-kernel credential structures.

These two features are not found in the FreeBSD MAC framework.

3. Implementation

1 Actually scope specific wrapper functions for convenience.

Prison Management

The central data structure for jails is the prison structure that records the attributes of a jail. When a process is being jailed a reference to a system wide prison structure is attached to the process' credentials. And the presence of a reference to a prison structure in the process credentials indicates that a process is jailed. We use *kauth_set_data(9)* to associate a prison reference to the process credentials structure. This is transparent to all other users of credentials. As the credential structure itself hasn't changed no other source code of the system has to be recompiled. In FreeBSD the credentials structure was changed to add a field with a pointer to the prison structure.

In NetBSD the attributes of a prison are passed into the kernel in the form of a *proplib(3)* plist dictionary to the *gaol(2)* system call. [PROPLIB3] This was done to allow future extensibility. In particular, for allowing to associate multiple network addresses with a jail.

Prison structures need to be reference counted and disposed of when the last credential structure referencing a prison is no longer in use. Fortunately, the kauth(9) framework provides a nice infrastructure for just this purpose. The gaol security model registers a listener for the credentials scope and is henceforth notified whenever a credential structure is initialized, copied, freed, or the referencing process forks. For copies and forks the prison's reference count is increased, and for frees the reference count is decreased, automatically disposing of the prison structure when the reference count reaches zero.

Jail-specific System Calls

The system call interface consists of the two system calls *goal(2)* and *gaol_attach()* which are closely modeled after the FreeBSD system calls *jail(2)* and *jail_attach(2)*. The difference is that *gaol(2)* takes a *proplib(3)* plist dictionary instead of a jail structure as argument. We chose different names for the system calls to make it obvious which interface version is used.

The kernel side of the implementation is boringly similar between NetBSD and FreeBSD. There are minor differences in internal kernel interfaces that need to be taken into account.

The biggest difference is that for *gaol(2)* the *proplib(3)* dictionary containing the jail arguments needs to be transfered into the kernel. For some strange reason the *proplib(3)* interface provides standardized methods to pass *proplib(3)* objects in and out of the kernel with *ioctl(2)* system calls, but there is no support for passing *proplib(3)* objects to other, more general system calls. This is easily fixed with a bit of almost trivial refactoring.

Authorizing Requests - The Trivial Ones

As mentioned in the FreeBSD jails paper [KAWA2000] the authors had to carefully read the kernel source code and identify all the places where privileges where checked and modify those places to make them aware of jails.

With the introduction of the kauth(9) framework most of this was done already in the NetBSD kernel. There were only a handful of places left that need to be made jail-aware but had no corresponding kauth(9) checks. These are described later.

Authorization decisions are made by so-called listeners for specific scopes. The listeners for the goal security model have a common code pattern:

- If the credentials do not have a reference to a prison, they punt the decision by returning KAUTH_RESULT_DEFER.
- Otherwise the default return value is set to KAUTH_RESULT_DENY, indicating that the requested action should be denied, as is appropriate for most of the actions.
- Then individual actions that are not always denied are examined in detail and the return value is set to KAUTH_RESULT_ALLOW, if appropriate.

```
int
secmodel_gaol_system_listener(kauth_cred_t cred, kauth_action_t action,
    void *cookie, void *arg0, void *arg1, void *arg2, void *arg3)
{
      int result;

      /* if not in jail, defer */
      if (gaoled(cred) == NULL)
            return (KAUTH_RESULT_DEFER);

      result = KAUTH_RESULT_DENY;

      switch (action) {
      case KAUTH_SYSTEM_ACCOUNTING:
      ...
      default:
            break;
      }

      return (result);
}
```

This covers the overwhelming majority of all requests because we deny most of them for imprisoned processes.

Authorizing Requests - The Easy Ones

The next class of authorization requests are the ones that are allowed unconditionally for imprisoned processes: e.g. *chroot(2)*. They are handled in the big switch statement of the listener function and set the result value to KAUTH_RESULT_DEFER.

```
      ...
      case KAUTH_SYSTEM_CHROOT:
            result = KAUTH_RESULT_DEFER;
            break;
      ...
```

There is also a class of privileged actions that are allowed inside jails depending on the setting of a sysctl variable: e.g. changing the system file flags. They are handled in the code of the listener functions similar to the unconditionally allowed ones by setting the result value to KAUTH_RESULT_DEFER when appropriate.

```
      ...
      KAUTH_SYSTEM_CHSYSFLAGS:
            if (gaol_chflags_allowed)
                  result = KAUTH_RESULT_DEFER;
            break;
```

```
        ...
```

The next class of authorization actions are the ones that require actual checks of prison attributes, e.g. the ones that ensure that processes from one prison can't see or interfere with processes in a different prison or ones that are not imprisoned.

```
        ...
        case KAUTH_PROCESS_CANSEE:
              result = prison_check(cred, ((struct proc *)arg0)->p_cred);
              break;
        ...
```

The most complicated of these checks is the check done upon opening a socket that restricts new sockets to PF_LOCAL, PF_INET, and PF_ROUTE protocol families. However, all the required information was already passed to the scope listener and the implementation is straight forward.

Authorization Requests - The Interesting Ones

Thus far implementing the gaol security model was almost boringly easy. Now, we come to the more interesting parts that are not as easy to implement.

Implementing these checks requires externally visible changes to the kauth(9) framework to be made. New authorization actions and sub-requests need to be defined. Because of this the enums detailing the authorization actions and sub-requests in the affected scopes need to be extended with new codes for the new actions.

This changes the binary interface of the kauth(9) KBI. Therefore it needs to be centrally coordinated. All existing security models need to be carefully checked whether they need to be made aware of the new authorization actions. Typically, at least the default bsd44 security model needs to allow these requests.

Unfortunately the kauth(9) framework has currently no means to check that a given security model is up-to-date with all the defined authorization actions. Thus it is possible to e.g. load an incompatible security model as an LKM.

Two new actions needed to be defined in the system scope. One authorizes the use of System V IPC system calls. Whether requests are denied or deferred depends on a sysctl variable that controls the behaviour for all jails. The other authorizes the use of the quotactl(2) system call and is denied for imprisoned processes. This is different from requiring super user privilege for certain quota operations.

The processes scope was extended with an authorization action to determine whether the calling process can wait(2) for a named processes. The request is denied if the named process is outside the prison of the requesting processes.

The network scope was extended with a request to check whether a given credential holder is allowed to see a particular interface address. Imprisoned processes can only see interface addresses that are specified when the prison is created.

When an imprisoned process binds a socket to a local address and when using raw sockets, which is optionally allowed, the kernel needs to check that the local address specified or embedded in the packets is on the list of allowed local network addresses for the prison. This is implemented in a straight forward manner with the new KAUTH_NETWORK_ADDR_CANSEE action in the network scope.

Because these new requests fit well with the kauth usage model the necessary changes are straight forward. On the calling sites standard code for the checks is added. The listeners in the system's default scope have to be modified to unconditionally allow these requests. All other security models have to be carefully checked if they need updating so that these requests are not erroneously denied.

What remains are a couple of special cases that at first glance don't seem to fit well into the kauth model. They are all related to replacing certain network addresses with an address from the list of local network addresses permitted in the prison.

The first case is in the routing socket code where interface addresses are returned to userland. FreeBSD returns only the prison's IP address for imprisoned processes. Implementing this with the kauth(9) framework requires to deviate from the purist model of only returning a go/no-go response from a request to authorize an action. Never letting ones sense of morals getting in the way of doing what is right, we extend the semantics of *kauth_authorize_action()* in the case of KAUTH_NETWORK_IFADDR so that “allow” means: “use the interface address as you normally would”. In the “deny” case, an alternate address that is to be used instead of the original interface address is passed back to the calling site through one of the optional parameters of *kauth_authorize_action()*.

Similarly, when an imprisoned process requests to bind a socket to the unspecified or loopback address or tries to connect to the loopback address, the jail code needs to fix up the address with an address from the list of local network addresses permitted in the jail. This is done through the KAUTH_NETWORK_LOCALADDR authorization action. The return value is ignored as it cannot fail. Instead the action simply fixes up the socket address that is passed in as one of the parameters.

The last case is sending packets through a raw socket. Again, the kernel needs to fix up the source address of outgoing packets with an address from the list of allowed local addresses for the prison. This is accomplished by another newly introduced authorization action in the network scope that returns a suitable local address through one of the parameters of *kauth_authorize_action()*.

4. Limitations

FreeBSD jails limit the visibility of sockets that belong to different jails. This has not yet been implemented. Emulating the FreeBSD *cr_canseesocket()* in the kauth(9) framework is just another trivial variant of the KAUTH_XXX_CANSEE actions. But it requires changes in the kernel infrastructure to attach full credentials to sockets. Due to time constraints this hasn't been looked at yet.

When this is implemented it could be reused by the curtain security model to control visibility of sockets with credentials holding different uids/gids.

Setting and testing a jail-specific securelevel has not been implemented.

Changing the jail-specific hostname from within the prison has not been implemented.

Shoehorning these two actions into the kauth(9) framework is challenging at best. The kauth(9) framework always invokes all listeners for a scope. However, the order in which they are invoked is not defined. One approach might be to adapt the techniques from the secmodel_overlay example.

In FreeBSD the ddb kernel debugger's *ps* command indicates indicates jailed processes with the flag letter 'J'. This has not been implemented.

FreeBSD's procfs indicates the jail's hostname in the per-process status file. This has not been implemented either.

5. Conclusions

Implementing jails within the kauth(9) framework as a proof-of-concept was surprisingly easy. However there remain a couple of points that are not entirely satisfactory. In particular the requests that need to chcange the network addresses at the calling sites makes assumptions that no other security model does the same. While this works out OK in practice, it is not safe under all theoretically possible scenarios. How this might be solved in a production-quality implementation requires further thought.

It is feasible to introduce new authorization scopes, because the scope identifiers are dynamically assigned. Adding new authorization actions or action-specific sub-requests, however, needs careful planning and central coordination, because of their implementation as enums. Also the lack of kauth(9) ABI versioning might lead to difficulties.

References

[KAWA2000]	Poul-Henning Kamp and Robert Watson, Jails: Confining the omnipotent root. http://phk.freebsd.dk/pubs/sane2000-jail.pdf
[TN2127]	Apple Computer Inc. Technical Note TN2127 Kernel Authorization http://developer.apple.com/technotes/tn2005/tn2127.html
[EFRAD2006]	Elad Efrad, NetBSD Security Enhancements, EuroBSDCon 2006, http://www.netbsd.org/~elad/recent/recent06.pdf
[KAUTH9]	kauth(9) man-page, kauth – kernel authorization framework http://netbsd.gw.com/cgi-bin/man-cgi?kauth++NetBSD-4.0
[PROPLIB3]	proplib(3) man-page, proplib – property container object library http://netbsd.gw.com/cgi-bin/man-cgi?proplib++NetBSD-4.0