Socket Filters

Anders Persson
anders.persson@sun.com

Version 1.2
January 19, 2010

Contents

1 Overview **2**

2 Filter Configuration **3**
2.1 Administrative Model 3
2.2 Changes to `soconfig(1M)` 3
2.3 Changes to the `sockconfig()` System Call 4
2.4 Zone Considerations 4

3 Attach Semantics **4**
3.1 Automatic Filters 5
3.2 Programmatic Filters 6
3.3 Passively Opened Sockets 7

4 Callbacks **7**
4.1 Execution Order 7
4.2 Concurrency 8
4.3 Stopping Callbacks 9

5 Filter Actions **9**
5.1 Injecting Data 9
5.2 Deferring Notifications of New Connections 9
5.3 Flow Control 9

6 Observability **10**
6.1 pfiles(1M) 10
6.2 truss(1M) 10
6.3 kstats 10

7 Filter Implications **11**
7.1 sendfile(3) 11
7.2 I/OAT 11
7.3 STREAMS Fallback 11

8 Socket Filter API **11**
8.1 Module Registration 11
8.2 Callback Functions 12
8.2.1 sofop_attach_active 12
8.2.2 sofop_attach_passive 12
8.2.3 sofop_detach 14
8.2.4 sofop_data_in 14
8.2.5 sofop_data_in_proc 14
8.2.6 sofop_data_out 15
8.2.7 sofop_bind and other Control Operations 15
8.2.8 sofop_mblk_prop 15
8.2.9 sofop_notify 16
8.3 Functions for Injecting Data 16
8.4 Other Functions 17
8.4.1 sof_bypass 17
8.4.2 sof_newconn_ready 17
8.4.3 sof_snd_flowctrl 17
8.4.4 sof_rcv_flowctrl 17
8.4.5 sof_get_cookie 18
8.4.6 sof_cas_cookie 18

1 Overview

A filter is configured to work with one or more non-STREAM sockets and it can be attached either automatically to all matching sockets or programmatically to a specific socket by the request of an application. Once attached, the filter exists in the socket layer, but sits logically in between the socket

and transport layers where it is notified of user requests and transport events via callback functions.

The action a filter can take depend on the callback. But typically a filter can modify or deny socket operations, transform, delay and inject data, as well as defer the notification of new connections.

2 Filter Configuration

2.1 Administrative Model

Filters are managed by `smf(5)` where each filter is represented by a separate service. Enabling the service registers the filter and makes it available for applications to use and disabling the service will make the filter immediately unavailable for new sockets, however, sockets already using the filter will continue doing so. All filter services have FMRIs that start with `svc:/network/socket-filter`, so, for example, a filter identified by "accf" would have an accompanying service named `svc:/network/socket-filter/accf:default`.

2.2 Changes to soconfig(1M)

The SMF services use `soconfig(1M)` to configure socket filters. When a filter is configured, it must provide a list of <family, type, proto>-tuples which it is interested in, and the attach semantics of the filter. For automatic filters, an additional placement hint can be specified (the impact of which is described in Section 3.1). Unconfiguring the filter is done by specifying the name while leaving out the other parameters.

```
soconfig -F name [modname {auto [top | bottom | before:filter |
     after:filter] | prog} family:type:protocol,...]
```

So, for example, to configure "accf", which can be programmatically attached to TCP sockets and is implemented in "accf_mod":

```
# soconfig -F accf accf_mod prog 2:2:0,2:2:6
```

The associated kernel module is not loaded until the filter is being attached to a socket.

To unconfigure the "accf" filter:

```
# soconfig -F accf
```

Which would make the filter unavailable for any new sockets being created, however, the filter will not be removed from sockets it has already been attached to.

The changes to `soconfig(1M)` will be *Consolidation-Private* and will not be documented in the man page.

2.3 Changes to the `sockconfig()` System Call

`soconfig(1M)` uses the `sockconfig()` system call to update the socket configuration. The system call will be changed so that is can handle both socket and filter configuration.

2.4 Zone Considerations

Socket configuration is done on system-wide basis and is only allowed to be changed from the global zone. Filters are tightly coupled with the socket configuration, so they too will only be managed on a system-wide basis. Filter services started in a non-global zone will transition to maintenance.

Since `sockconfig()` can only be issued in the global zone, no system call emulation is needed for solaris10 branded zones.

3 Attach Semantics

A filter can only attach once to a specific socket, but there may be multiple filters attached to the same socket, which result in a *filter stack* (Figure 1). The stack is built bottom up, so if a filter F1 attached before F2, then F1 would exist below F2 in the stack (i.e. F1 is closer to the transport).

The way in which a filter is attached to a socket differs for actively opened (via `socket()`) and passively opened (via `accept()`) sockets. In the case of an active open, the method in which a filter is attached depends on whether it is an automatic or programmatic filter, whereas for passively opened sockets the filters are simply inherited from the parent.

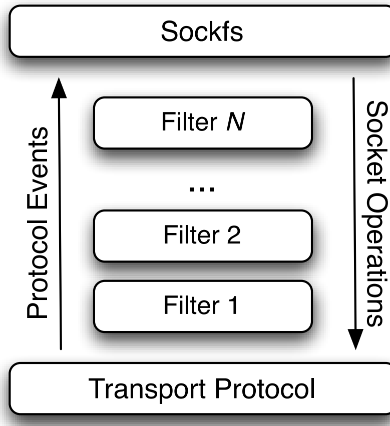

Figure 1: Socket Filter Stack

3.1 Automatic Filters

Automatic filters are meant to be transparent to the application and can attach without any intervention from the application. When `socket()` is called, the kernel will determine whether there are any automatic filters that have been configured for the socket type being created and each such filter has the opportunity to attach. In the event that a filter wants to attach, but is unable to do so (for example, if it fails to allocate private data), then it can either force the `socket()` call to fail, or it could let the creation of the socket continue by indicating that it does not want to attach to the socket. The filter can use the same mechanism to actively deny certain sockets from being created.

The order in which automatic filters are attached depend on whether any placement hints were specified (which was adopted from PFHooks [1]). Four types of hints are available:

`top` Indicates that the filter wants to be the last *automatic* filter to attach to the socket (i.e., it wants to be closest to the socket). For a given socket type, there can only be one filter with the hint.

`bottom` Indicates that the filter wants to be the first *automatic* filter to attach to the socket (i.e., it wants to be closest to the protocol). For a given socket type, there can only be one filter with the hint.

`before:name` Indicates that the filter wants to attach before the specified *automatic* filter (i.e., it wants to be closer to the protocol).

`after:name` Indicates that the filter wants to attach after the specified *automatic* filter (i.e., it wants to be closer to the socket).

Automatic filters will always attach before any programmatic filters, so they will always be lower in the stack.

3.2 Programmatic Filters

Programmatic filters are controlled using `setsockopt()` and `getsockopt()`. This project introduces three new *Consolidation Private* socket options at a new option level, `SOL_FILTER`:

`FIL_ATTACH`: `setsockopt` specific option used to attach a *programmatic* filter to the socket. The application must specify the name of the filter it wants to attach. The operation fails with `EEXIST` if the specified filter is already attached, and `ENXIO` if it is not available.

`FIL_DETACH`: `setsockopt` specific option used to detach a *programmatic* filter from the socket. The application must specify the name of the filter that should be detached. An error (`ENXIO`) is returned if the filter is not attached.

`FIL_LIST`: `getsockopt` specific option to retrieve a list of filters (both automatic and programmatic) that are attached to the socket. The application must specify a buffer large enough to contain at least one `struct fil_info` (Table 1), and `EINVAL` is returned otherwise. The filter stack information is retrieved from top to bottom, so the `fi_pos` field of the first entry can be used to determine how many filters are attached and whether the buffer was large enough.

Both `FIL_ATTACH` and `FIL_DETACH` are only allowed as long a no other socket operations (`bind()`, `send()`, etc.) have been issued on the socket (and an attach/detach attempt after that will fail with `EINVAL`). By restricting the attach period, a filter is guaranteed that the socket is in a known state when it is attached.

It is also important to point out that automatic filters cannot be controlled using `setsockopt()`. The idea of adding a policies to indicate whether

```
/*
 * Structure returned by FIL_LIST
 */
struct fil_info {
        int     fi_flags;               /* see below (FILF_*) */
        int     fi_pos;                 /* position (0 is bottom) */
        char    fi_name[FILNAME_MAX];   /* filter name */
};

#define FILF_PROG       0x1             /* programmatic attach */
#define FILF_AUTO       0x2             /* automatic attach */
#define FILF_BYPASS     0x4             /* filter is not active */
```

Table 1: Structure for retrieving filter information

an automatic filter could be controlled in this way was considered, but was not deemed necessary at this time.

3.3 Passively Opened Sockets

When a new connection comes in, and a socket is created, each filter that is attached to the parent socket has an opportunity to attach to the new socket. If a filter fails to attach, then it can allow the socket creation to continue (but of course, the filter will not be attached), or it can deny the connection.

It is not possible to attach any additional programmatic filters to a socket once it has been accepted. Therefore, an application must attach all filters of interest to the listening socket. So the filters attached to a passively opened socket will always be a subset of the filters attached to the parent.

4 Callbacks

The available callbacks are shown in Table 2.

4.1 Execution Order

In the case of multiple filters, the execution order of the callback functions is determined by the order of the filter in the stack and the source of the

Callbacks	Direction
`bind`, `listen`, `connect`, `getsockopt`, `setsockopt`, `getsockname`, `getpeername`, `shutdown`, `ioctl`, `data_out`, `detach`	$\downarrow$
`data_in`, `data_in_proc`	$\uparrow$
`attach`, `notify`	$\uparrow\downarrow$

Table 2: Callbacks and how they traverse the filter stack

event. The direction in which a callback traverse the stack is shown in Table 2. In general, socket operations (`connect()`, `bind()`, etc.) traverse the stack top to bottom, while protocol events traverse the stack bottom-up. However, there are a couple of exceptions that are worth pointing out:

`data_in_proc` is called from the application's context when it issues a `read()` and allows the filter to process the data before it is copied out, and is meant for data transformations that are too expensive to do when the protocol delivers the data. Although this callback is triggered by the application, it actually traverses the stack bottom-up to ensure that inverse transformations are applied in the right order.

`notify, attach` can be called from both the socket and protocol layers, and the direction of traversal is determined by the source.

4.2 Concurrency

With the exception of `attach` and `detach`, callbacks can execute simultaneously, so in general filters must be MP-safe. However, in certain situations the properties of a transport protocol will ensure that events will be serialized. For example, `SOCK_STREAM` sockets ensure that data arrives as an in-order data stream, and therefore the `data_in` callback will be serialized. A filter must guarantee to uphold the properties of the protocol. So in the `SOCK_STREAM` example, if the filter ends up queuing data locally, then it must ensure that when the data is finally delivered, it is done so in the same order it was originally received.

4.3 Stopping Callbacks

Although a filters cannot be detached when the socket is active, a "bypass" flag can be set on a filter such that it will stop receiving callbacks. However, there is no guarantee that there are no active callbacks after the flag has been set, only that no new callbacks will be triggered. A typical use of the "bypass" flag would be a filter that only needs to be active during a certain period of a sockets lifetime. It should be noted that setting the flag is just a performance enhancement that can be used to minimize the impact of a filter when it is no longer is interested in socket events.

5 Filter Actions

5.1 Injecting Data

Filters can asynchronously enqueue data on the socket's receive queue or send out data to its peer. When a filter injects data out, only filters below it will observe the data (which will appear like a normal `data_out` event). Similarly, when a filter injects data in to the socket, filters above it in the stack will observe an `data_in` event.

5.2 Deferring Notifications of New Connections

When a filter is attached to a passively opened socket, it can force the notification of the new socket to be deferred. The connection can be deferred by multiple filters, and only when all filters have marked the connection as "ready" will the notification take place.

A deferred connection is still accounted for in the accept queue so one potential problem that could occur is that the accept queue becomes full of deferred connections. In such an event, the framework will ensure that new sockets can still be enqueued.

5.3 Flow Control

A filter can enable flow control on both the receive and transmit sides. In case multiple filters, then flow control will only be lifted once all filters have cleared the flow control condition.

Counter	Description
`nactive`	Num. to sockets the filter is currently attached to
`ndeferred`	Num. of sockets currently deferred by filter
`tot_active_attach`	Num of actively opened sockets the filter has attached to
`tot_passive_attach`	Num. of passively opened sockets the filter has attached to

Table 3: Filter kstat, sockfs::filter_<filtername>

6 Observability

6.1 pfiles(1M)

`pfiles(1M)` will be updated to include filter information when the file descriptor is a socket. In the example output below, the socket has two filters, one automatic and one programmatic. The `::pfiles` mdb command will also be updated to provide identical information.

```
3: S_IFSOCK mode:0666 dev:379,0 ino:61145 uid:0 gid:0 size:0
   O_RDWR
     SOCK_STREAM
     SO_SNDBUF(49152),SO_RCVBUF(49152)
     filters: accf(prog), logf(auto)
     sockname: AF_INET 0.0.0.0  port: 0
```

6.2 truss(1M)

`truss(1M)` will be updated to understand the new subcodes for the `sockconfig()` system call.

6.3 kstats

Each filter will have a kstat entry named `sockfs::filter_<filtername>` (Table 3) and there is one global kstat, `sockfs::sockfilter` (Table 4).

Counter	Description
`defer_closed`	Num. of deferred connections that have been closed
`defer_close_backlog`	Num. of deferred connections about to be closed
`defer_close_failed`	Num of deferred connections that could not be closed

Table 4: Global kstat, sockfs::sockfilter

7 Filter Implications

7.1 `sendfile(3)`

`sendfile()` frequently uses segmap to transmit memory mapped mblks which should not be modified. To avoid unnecessary work (i.e., copymsg(9F) the segmapped mblk), segmap will not be used if an attached filter has a `data_out` callback.

7.2 I/OAT

Intel I/OAT allows a copyout operation to start asynchronously as the data arrives at the socket. However, a filter may need to transform data before making it available to the application, in which case I/OAT cannot be used. As a result, I/OAT will be disabled if the filter has a `data_in_proc` callback.

7.3 STREAMS Fallback

A socket will not be able to fallback to STREAMS if the socket has any filter that has not set the "bypass" flag (see Section 4.3). The restriction is made to ensure that data that arrives at the socket is not garbled, which it otherwise might be if a filter needs to process it (e.g., decrpyt or uncompress).

8 Socket Filter API

The Socket Filter API will be Consolidation-Private.

8.1 Module Registration

When `_init()` is called, the module is expected to call `sof_register()` to register the module. Similarly, when `_fini()` is called the module should

try to unregister by calling `sof_unregister()`.

```
int sof_register(int version, const char *name, sof_ops_t *ops,
    int flags);

int sof_unregister(const char *name);
```

When calling `sof_register()`, the module should specify the current version by passing in the constant `SOF_VERSION`, the name of the filter, and a set of callback functions. The "flags" argument is currently unused and should be 0. When unregistering, only the name of the filter needs to be specified. In both cases an error is indicated by a non-zero return value.

8.2 Callback Functions

The filter can specify a number of callbacks (Table 5), most of which have a return value of `sof_rval_t` (Table 6). If a filter uses asynchronous operations, then it *must* specify a `sofop_notify` callback (details in Section 8.2.9).

8.2.1 `sofop_attach_active`

```
typedef sof_rval_t  (*sof_attach_active_fn_t)(sof_handle_t h, int family,
    int type, int proto, cred_t *credp, void **cookiep);
```

`sofop_attach_active` is called when the filter is being attached to an actively opened socket. The filter can return `SOF_RVAL_DETACH` if the filter is not interested in the socket and wants to be removed from it. The filter is not allowed to return `SOF_RVAL_RETURN` or call any of the functions described in sections 8.3 and 8.4.

8.2.2 `sofop_attach_passive`

```
typedef sof_rval_t  (*sof_attach_passive_fn_t)(sof_handle_t h,
    sof_handle_t parent, void *pcookie, struct sockaddr *laddr,
    socklen_t laddrlen, struct sockaddr *faddr,
    socklen_t faddrlen, void **cookiep);
```

```
typedef struct sof_ops_s {
        sof_attach_active_fn_t  sofop_attach_active;
        sof_attach_passive_fn_t sofop_attach_passive;
        sof_detach_fn_t         sofop_detach;
        sof_data_in_fn_t        sofop_data_in;
        sof_data_in_proc_fn_t   sofop_data_in_proc;
        sof_data_out_fn_t       sofop_data_out;
        sof_bind_fn_t           sofop_bind;
        sof_listen_fn_t         sofop_listen;
        sof_connect_fn_t        sofop_connect;
        sof_accept_fn_t         sofop_accept;
        sof_getsockopt_fn_t     sofop_getsockopt;
        sof_setsockopt_fn_t     sofop_setsockopt;
        sof_getsockname_fn_t    sofop_getsockname;
        sof_getpeername_fn_t    sofop_getpeername;
        sof_shutdown_fn_t       sofop_shutdown;
        sof_ioctl_fn_t          sofop_ioctl;
        sof_mblk_prop_fn_t      sofop_mblk_prop;
        sof_notify_fn_t         sofop_notify;
} sof_ops_t;
```

Table 5: Filter Callbacks

Value	Impact
`SOF_RVAL_CONTINUE`	Continue normal processing
`SOF_RVAL_RETURN`	Causes the operation to return immediately but without an error
`SOF_RVAL_DETACH`	Detach the filter (`sofop_attach_*` specific)
`SOF_RVAL_DEFER`	Defers the notification of a new connection (`sofop_attach_passive` specific)
`SOF_RVAL_EINVAL`	Stops the operation, returning `EINVAL`
`SOF_RVAL_EACCES`	Stops the operation, returning `EACCES`
`SOF_RVAL_ENOMEM`	Stops the operation, returning `ENOMEM`
`SOF_RVAL_ECONNABORTED`	Stops the operation, returning `ECONNABORTED`

Table 6: Values of `sof_rval_t` and their impact

sofop_attach_passive is called when the filter is being attached to an passively opened socket. The filter can return SOF_RVAL_DETACH if the filter is not interested in the socket and wants to be removed from it, and the notification of the connection can be deferred by returning SOF_RVAL_DEFER. The filter is not allowed to return SOF_RVAL_RETURN or call any of the functions described in sections 8.3 and 8.4.

8.2.3 sofop_detach

```
typedef void  (*sof_detach_fn_t)(sof_handle_t h, void *cookie,
    cred_t *credp);
```

sofop_detach is called when the filter is being detached and must result in the filter stopping all asynchronous operations and freeing any resources associated with the socket. The filter is *not* allowed to use any of the functions in sections 8.3 and 8.4 from this callback.

8.2.4 sofop_data_in

```
typedef void (*sof_data_in_fn_t)(sof_handle_t h, void *cookie,
    mblk_t **mpp, size_t *sizep, int *flagsp);
typedef mblk_t *(*sof_data_in_fn_t)(sof_handle_t h, void *cookie,
    mblk_t *mp, size_t *sizep);
```

sofop_data_in is called when the data arrives from the protocol. The filter can modify, free or replace the mblk. If the size of the data is changed, then "sizep" must be updated accordingly. The "flagsp" argument contains additional information regarding the data, for example, MSG_OOB. Since this callback is on the protocol data path, the filter should keep any processing to a minimum to ensure that performance is not degraded. Any time consuming operations should be deferred to sofop_data_in_proc.

The callback should return a mblk ready to be enqueued on the socket buffer, or NULL if the mblk was consumed or an error was encountered. In case of an error, the filter must free the mblk.

8.2.5 sofop_data_in_proc

```
typedef mblk_t *(*sof_data_in_proc_fn_t)(sof_handle_t h, void *cookie,
    mblk_t *mp, size_t *sizep);
```

`sofop_data_in_proc` is called before data is copied out. The "mp" is the mblk being processed. The filter is allowed to modify, free or replace the mblk, but any change in size must result in "sizep" being updated. The callback should return a mblk that is ready to be copied to the application, or NULL if the data was consumed or an error occurred. In case of error, the mblk should be freed by the filter. The callback will only be triggered once for each mblk in the receive queue.

8.2.6 `sofop_data_out`

```
typedef mblk_t *(*sof_data_out_fn_t)(sof_handle_t h, void *cookie,
    struct nmsghdr *msg, mblk_t *mp, size_t *sizep, sof_rval_t *rvalp);
```

`sofop_data_out` is called before data is sent down to the protocol. The filter is allowed to change the message header and mblk, but any changes to the size of the mblk must result in "sizep" being updated. The send operation will continue if the callback returns an mblk. If the filter consumes the original mblk or if an error is encoutered, NULL should be returned and "rvalp" updated. The filter needs to free mblk in case of error.

8.2.7 `sofop_bind` and other Control Operations

The semantics of the callbacks associated with control operations (`bind()`, `connect()`, `setsockopt()`, etc.) are the same, and here we only look at `sofop_bind` as an representative for all of them.

```
typedef sof_rval_t (*sof_bind_fn_t)(sof_handle_t h, void *cookie,
    struct sockaddr *addr, socklen_t addrlen, cred_t *credp);
```

`sofop_bind` is called before the bind operation is issued to the protocol and the filter is allowed to modify "addr". It is possible for the filter to deny the operation by returning `SOF_RVAL_E*`, or cause an immediate return (without error) by returning `SOF_RVAL_RETURN`.

8.2.8 `sofop_mblk_prop`

```
typedef void (*sof_mblk_prop_fn_t)(sof_handle_t h, void *cookie,
    ssize_t *maxblk, ushort_t *wroff, ushort_t *tail);
```

Event	Argument
SOF_EV_CLOSING	OK to inject data? (boolean)
SOF_EV_INJECT_DATA_IN_OK	None
SOF_EV_INJECT_DATA_OUT_OK	None
SOF_EV_CONNECTED	None
SOF_EV_CONNECTFAILED	Error code (int)
SOF_EV_DISCONNECTED	Error code (int)
SOF_EV_CANTRECVMORE	None
SOF_EV_CANTSENDMORE	None

Table 7: Values for `sof_event_t`

The mblk properties control the size, header and tail space for mblks being sent by the socket. `sofop_mblk_prop` is called when one or more of the mblk properties for the socket has changed and allows the filter to modify the properties. However, care must be taken if the filter increases "maxblk" or decreases "wroff" or "tail"; it must then ensure that outgoing mblks satisfy the original properties before passing it along.

8.2.9 `sofop_notify`

```
typedef void (*sof_notify_fn_t)(sof_handle_t h, void *cookie,
    sof_event_t event, void *arg);
```

`sofop_notify` is called when the status of the socket changes. Table 7 lists the possible events how the "arg" should be interpreted. When a filter receives a `SOF_EV_CLOSING` notification, then it *must* cancel any pending asynchronous operations and timers before returning. It is possible for other callbacks to be triggered in the period between the `notify` and `detach` callbacks. For example, data might arrive from the protocol. The filter is allowed to act on those event as long as it is done synchronously.

8.3 Functions for Injecting Data

```
int sof_inject_data_in(sof_handle_t h, mblk_t *mp, size_t len,
    int flags, boolean_t *flowctrl);

int sof_inject_data_out(sof_handle_t h, mblk_t *mp,
    struct nmsghdr *msg, boolean_t *flowctrld);
```

sof_inject_data_in and sof_inject_data_out can be used by the filter to send data to the socket or peer. When data is successfully injected, zero is returned; otherwise an errno is returned. Flow control is not enforced, however, "flowctrld" will be set to TRUE if the socket is flow controlled, in which case the filter should stop injecting data until it gets notified (via sofop_notify) that it's OK to resume.

Neither of these two functions are allowed to be called from sofop_attach and sofop_detach.

8.4 Other Functions

8.4.1 sof_bypass

```
void sof_bypass(sof_handle_t h);
```

sof_bypass Sets the "bypass" flag on the filter as described in Section 4.3. Once set, it can not be removed.

8.4.2 sof_newconn_ready

```
void sof_newconn_ready(sof_handle_t h);
```

sof_newconn_ready is called by a filter that has deferred the notification of a new connection and which now wants to make the connection available.

8.4.3 sof_snd_flowctrl

```
void    sof_snd_flowctrl(sof_handle_t h, boolean_t enable);
```

sof_snd_flowctrl allows the filter to enable flow control on the transmit side of the socket.

8.4.4 sof_rcv_flowctrl

```
void    sof_rcv_flowctrl(sof_handle_t h, boolean_t enable);
```

sof_rcv_flowctrl allows the filter to enable flow control on the receive side of the socket.

8.4.5 `sof_get_cookie`

```
void *sof_get_cookie(sof_handle_t h);
```

`sof_get_cookie` returns the cookie used by “h”.

8.4.6 `sof_cas_cookie`

```
void *sof_cas_cookie(sof_handle_t h, void *old, void *new);
```

`sof_cas_cookie` does a compare-and-swap of the cookie used by “h”.

References

[1] PSARC/2008/219 Committed API for packet interception, http://arc.opensolaris.org/caselog/PSARC/2008/219/