Subject: Re: [RFC] network namespaces
Posted by serue on Wed, 16 Aug 2006 11:53:13 GMT
View Forum Message <> Reply to Message

Quoting Andrey Savochkin (saw@sw.ru):
> Hi All,
>
> I'd like to resurrect our discussion about network namespaces.
> In our previous discussions it appeared that we have rather polar concepts
> which seemed hard to reconcile.
> Now I have an idea how to look at all discussed concepts to enable everyone's
> usage scenario.
>
> 1. The most straightforward concept is complete separation of namespaces,
>   covering device list, routing tables, netfilter tables, socket hashes, and
>   everything else.
>
>   On input path, each packet is tagged with namespace right from the
>   place where it appears from a device, and is processed by each layer
>   in the context of this namespace.
>   Non-root namespaces communicate with the outside world in two ways: by
>   owning hardware devices, or receiving packets forwarded them by their parent
>   namespace via pass-through device.
>
>   This complete separation of namespaces is very useful for at least two
>   purposes:
>   - allowing users to create and manage by their own various tunnels and
>   VPNs, and
>   - enabling easier and more straightforward live migration of groups of
>   processes with their environment.

I conceptually prefer this approach, but I seem to recall there were
actual problems in using this for checkpoint/restart of lightweight
(application) containers. Performance aside, are there any reasons why
this approach would be problematic for c/r?

I'm afraid Daniel may be on vacation, and don't know who else other than
Eric might have thoughts on this.

> 2. People expressed concerns that complete separation of namespaces
>   may introduce an undesired overhead in certain usage scenarios.
>   The overhead comes from packets traversing input path, then output path,
>   then input path again in the destination namespace if root namespace
>   acts as a router.
>
>   So, we may introduce short-cuts, when input packet starts to be processes
>   in one namespace, but changes it at some upper layer.
>   The places where packet can change namespace are, for example:

>   routing, post-routing netfilter hook, or even lookup in socket hash.
>
>   The cleanest example among them is post-routing netfilter hook.
>   Tagging of input packets there means that the packets is checked against
>   root namespace's routing table, found to be local, and go directly to
>   the socket hash lookup in the destination namespace.
>   In this scheme the ability to change routing tables or netfilter rules on
>   a per-namespace basis is traded for lower overhead.
>
>   All other optimized schemes where input packets do not travel
>   input-output-input paths in general case may be viewed as short-cuts in
>   scheme (1). The remaining question is which exactly short-cuts make most
>   sense, and how to make them consistent from the interface point of view.
>
> My current idea is to reach some agreement on the basic concept, review
> patches, and then move on to implementing feasible short-cuts.
>
> Opinions?
>
> Next in this thread are patches introducing namespaces to device list,
> IPv4 routing, and socket hashes, and a pass-through device.
> Patches are against 2.6.18-rc4-mm1.

Just to provide the extreme other end of implementation options, here is
the bsdjail based version I've been using for some testing while waiting
for network namespaces to show up in -mm :)

(Not intended for *any* sort of inclusion consideration :)

Example usage:
 ifconfig eth0:0 192.168.1.16
 echo -n "ip 192.168.1.16" > /proc/$$/attr/exec
 exec /bin/sh

-serge

From: Serge E. Hallyn <hallyn@sergelap.(none)>
Date: Wed, 26 Jul 2006 21:47:13 -0500
Subject: [PATCH 1/1] bsdjail: define bsdjail lsm

Define the actual bsdjail LSM.

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
---
 security/Kconfig | 11
 security/Makefile | 1
 security/bsdjail.c | 1351 ++
 3 files changed, 1363 insertions(+), 0 deletions(-)

```
diff --git a/security/Kconfig b/security/Kconfig
index 67785df..fa30e40 100644
--- a/security/Kconfig
+++ b/security/Kconfig
@@ -105,6 +105,17 @@ config SECURITY_SECLVL

    If you are unsure how to answer this question, answer N.

+config SECURITY_BSDJAIL
+ tristate "BSD Jail LSM"
+ depends on SECURITY
+ select SECURITY_NETWORK
+ help
+   Provides BSD Jail compartmentalization functionality.
+   See Documentation/bsdjail.txt for more information and
+   usage instructions.
+
+   If you are unsure how to answer this question, answer N.
+
 source security/selinux/Kconfig

 endmenu
diff --git a/security/Makefile b/security/Makefile
index 8cbbf2f..050b588 100644
--- a/security/Makefile
+++ b/security/Makefile
@@ -17,3 +17,4 @@ obj-$(CONFIG_SECURITY_SELINUX)  += selin
 obj-$(CONFIG_SECURITY_CAPABILITIES) += commoncap.o capability.o
 obj-$(CONFIG_SECURITY_ROOTPLUG)  += commoncap.o root_plug.o
 obj-$(CONFIG_SECURITY_SECLVL)  += seclvl.o
+obj-$(CONFIG_SECURITY_BSDJAIL)  += bsdjail.o
diff --git a/security/bsdjail.c b/security/bsdjail.c
new file mode 100644
index 0000000..d800610
--- /dev/null
+++ b/security/bsdjail.c
@@ -0,0 +1,1351 @@
+/*
+ * File: linux/security/bsdjail.c
+ * Author: Serge Hallyn (serue@us.ibm.com)
+ * Date: Sep 12, 2004
+ *
+ * (See Documentation/bsdjail.txt for more information)
+ *
+ * Copyright (C) 2004 International Business Machines <serue@us.ibm.com>
+ *
+ *   This program is free software; you can redistribute it and/or modify
```

```
+ *   it under the terms of the GNU General Public License as published by
+ *   the Free Software Foundation; either version 2 of the License, or
+ *   (at your option) any later version.
+ */
+
+#include <linux/config.h>
+#include <linux/module.h>
+#include <linux/kernel.h>
+#include <linux/init.h>
+#include <linux/security.h>
+#include <linux/namei.h>
+#include <linux/namespace.h>
+#include <linux/proc_fs.h>
+#include <linux/in.h>
+#include <linux/in6.h>
+#include <linux/pagemap.h>
+#include <linux/ip.h>
+#include <net/ipv6.h>
+#include <linux/mount.h>
+#include <linux/netdevice.h>
+#include <linux/inetdevice.h>
+#include <linux/seq_file.h>
+#include <linux/un.h>
+#include <linux/smp_lock.h>
+#include <linux/kref.h>
+#include <asm/uaccess.h>
+
+static int jail_debug;
+module_param(jail_debug, int, 0);
+MODULE_PARM_DESC(jail_debug, "Print bsd jail debugging messages.\n");
+
+#define DBG 0
+#define WARN 1
+#define bsdj_debug(how, fmt, arg... ) \
+ do { \
+  if ( how || jail_debug ) \
+   printk(KERN_NOTICE "%s: %s: " fmt, \
+    MY_NAME, __FUNCTION__ , \
+    ## arg ); \
+ } while ( 0 )
+
+#define MY_NAME "bsdjail"
+
+/* flag to keep track of how we were registered */
+static int secondary;
+
+/*
+ * The task structure holding jail information.
```

```
+ * Taskp->security points to one of these (or is null).
+ * There is exactly one jail_struct for each jail.  If >1 process
+ * are in the same jail, they share the same jail_struct.
+ */
+struct jail_struct {
+ struct kref  kref;
+
+ /* these are set on writes to /proc/<pid>/attr/exec */
+ char *ip4_addr_name;  /* char * containing ip4 addr to use for jail */
+ char *ip6_addr_name;  /* char * containing ip6 addr to use for jail */
+
+ /* these are set when a jail becomes active */
+ __u32 addr4;       /* internal form of ip4_addr_name */
+ struct in6_addr addr6; /* internal form of ip6_addr_name */
+
+ /* Resource limits.  0 = no limit */
+ int max_nrtask;  /* maximum number of tasks within this jail. */
+ int cur_nrtask; /* current number of tasks within this jail. */
+ long maxtimeslice;      /* max timeslice in ms for procs in this jail */
+ long nice;       /* nice level for processes in this jail */
+ long max_data, max_memlock;  /* equivalent to RLIMIT_{DATA, MEMLOCK} */
+/* values for the jail_flags field */
+#define IN_USE 1  /* if 0, task is setting up jail, not yet in it */
+#define GOT_IPV4 2
+#define GOT_IPV6 4  /* if 0, ipv4, else ipv6 */
+ char jail_flags;
+};
+
+/*
+ * disable_jail:  A jail which was in use, but has no references
+ * left, is disabled - we free up the mountpoint and dentry, and
+ * give up our reference on the module.
+ *
+ *   don't need to put namespace, it will be done automatically
+ *     when the last process in jail is put.
+ *   DO need to put the dentry and vfsmount
+ */
+static void
+disable_jail(struct jail_struct *tsec)
+{
+ module_put(THIS_MODULE);
+}
+
+
+static void free_jail(struct jail_struct *tsec)
+{
+ if (!tsec)
+  return;
```

```
+
+ kfree(tsec->ip4_addr_name);
+ kfree(tsec->ip6_addr_name);
+ kfree(tsec);
+}
+
+/* release_jail:
+ * Callback for kref_put to use for releasing a jail when its
+ * last user exits.
+ */
+static void release_jail(struct kref *kref)
+{
+ struct jail_struct *tsec;
+
+ tsec = container_of(kref, struct jail_struct, kref);
+ disable_jail(tsec);
+ free_jail(tsec);
+}
+
+/*
+ * jail_task_free_security: this is the callback hooked into LSM.
+ * If there was no task->security field for bsdjail, do nothing.
+ * If there was, but it was never put into use, free the jail.
+ * If there was, and the jail is in use, then decrement the usage
+ *  count, and disable and free the jail if the usage count hits 0.
+ */
+static void jail_task_free_security(struct task_struct *task)
+{
+ struct jail_struct *tsec = task->security;
+
+ if (!tsec)
+  return;
+
+ if (!(tsec->jail_flags & IN_USE)) {
+  /*
+   * someone did 'echo -n x > /proc/<pid>/attr/exec' but
+   * then forked before execing.  Nuke the old info.
+   */
+  free_jail(tsec);
+  task->security = NULL;
+  return;
+ }
+ tsec->cur_nrtask--;
+ /* If this was the last process in the jail, delete the jail */
+ kref_put(&tsec->kref, release_jail);
+}
+
+static struct jail_struct *
```

```
+alloc_task_security(struct task_struct *tsk)
+{
+ struct jail_struct *tsec;
+
+ tsec = kmalloc(sizeof(struct jail_struct), GFP_KERNEL);
+ if (tsec) {
+  memset(tsec, 0, sizeof(struct jail_struct));
+  tsk->security = tsec;
+ }
+ return tsec;
+}
+
+static inline int
+in_jail(struct task_struct *t)
+{
+ struct jail_struct *tsec = t->security;
+
+ if (tsec && (tsec->jail_flags & IN_USE))
+  return 1;
+
+ return 0;
+}
+
+/*
+ * If a network address was passed into /proc/<pid>/attr/exec,
+ * then process in its jail will only be allowed to bind/listen
+ * to that address.
+ */
+static void
+setup_netaddress(struct jail_struct *tsec)
+{
+ unsigned int a, b, c, d, i;
+ unsigned int x[8];
+
+ tsec->jail_flags &= ~(GOT_IPV4 | GOT_IPV6);
+ tsec->addr4 = 0;
+ ipv6_addr_set(&tsec->addr6, 0, 0, 0, 0);
+
+ if (tsec->ip4_addr_name) {
+  if (sscanf(tsec->ip4_addr_name, "%u.%u.%u.%u",
+      &a, &b, &c, &d) != 4)
+   return;
+  if (a>255 || b>255 || c>255 || d>255)
+   return;
+  tsec->addr4 = htonl((a<<24) | (b<<16) | (c<<8) | d);
+  tsec->jail_flags |= GOT_IPV4;
+  bsdj_debug(DBG, "Network (ipv4) set up (%s)\n",
+   tsec->ip4_addr_name);
```

```
+ }
+
+ if (tsec->ip6_addr_name) {
+  if (sscanf(tsec->ip6_addr_name, "%x:%x:%x:%x:%x:%x:%x:%x",
+   &x[0], &x[1], &x[2], &x[3], &x[4], &x[5], &x[6],
+   &x[7]) != 8) {
+   printk(KERN_INFO "%s: bad ipv6 addr %s\n", __FUNCTION__,
+    tsec->ip6_addr_name);
+   return;
+  }
+  for (i=0; i<8; i++) {
+   if (x[i] > 65535) {
+    printk("%s: %x > 65535 at %d\n", __FUNCTION__, x[i], i);
+    return;
+   }
+   tsec->addr6.in6_u.u6_addr16[i] = htons(x[i]);
+  }
+  tsec->jail_flags |= GOT_IPV6;
+  bsdj_debug(DBG, "Network (ipv6) set up (%s)\n",
+   tsec->ip6_addr_name);
+ }
+}
+
+/*
+ * enable_jail:
+ * Called when a process is placed into a new jail to handle the
+ * actual creation of the jail.
+ *   Creates namespace
+ *   Stores the requested ip address
+ *   Registers a unique pseudo-proc filesystem for this jail
+ */
+static int enable_jail(struct task_struct *tsk)
+{
+ struct jail_struct *tsec = tsk->security;
+ int retval = -EFAULT;
+
+ if (!tsec)
+  goto out;
+
+ /* set up networking */
+ if (tsec->ip4_addr_name || tsec->ip6_addr_name)
+  setup_netaddress(tsec);
+
+ tsec->cur_nrtask = 1;
+ if (tsec->nice)
+  set_user_nice(current, tsec->nice);
+ if (tsec->max_data) {
+  current->signal->rlim[RLIMIT_DATA].rlim_cur = tsec->max_data;
```

```
+  current->signal->rlim[RLIMIT_DATA].rlim_max = tsec->max_data;
+ }
+ if (tsec->max_memlock) {
+  current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur =
+     tsec->max_memlock;
+  current->signal->rlim[RLIMIT_MEMLOCK].rlim_max =
+     tsec->max_memlock;
+ }
+ if (tsec->maxtimeslice) {
+  current->signal->rlim[RLIMIT_CPU].rlim_cur = tsec->maxtimeslice;
+  current->signal->rlim[RLIMIT_CPU].rlim_max = tsec->maxtimeslice;
+ }
+ /* success and end */
+ kref_init(&tsec->kref);
+ tsec->jail_flags |= IN_USE;
+
+ /* won't let ourselves be removed until this jail goes away */
+ try_module_get(THIS_MODULE);
+
+ return 0;
+
+out:
+ return retval;
+}
+
+/*
+ * LSM /proc/<pid>/attr hooks.
+ * You may write into /proc/<pid>/attr/exec:
+ *    lock  (no value, just to specify a jail)
+ *    ip 2.2.2.2
+ etc...
+ * These values will be used on the next exec() to set up your jail
+ *  (assuming you're not already in a jail)
+ */
+static int
+jail_setprocattr(struct task_struct *p, char *name, void *value, size_t rsize)
+{
+ struct jail_struct *tsec = current->security;
+ long val;
+ char *v = value;
+ int start, len;
+ size_t size = rsize;
+
+ if (tsec && (tsec->jail_flags & IN_USE))
+  return -EINVAL;  /* let them guess why */
+
+ if (p != current || strcmp(name, "exec"))
+  return -EPERM;
```

```
+
+ if (!tsec) {
+  tsec = alloc_task_security(current);
+  if (!tsec)
+   return -ENOMEM;
+ }
+
+ if (v[size-1] == '\n')
+  size--;
+
+ if (strncmp(value, "ip ", 3) == 0) {
+  kfree(tsec->ip4_addr_name);
+  start = 3;
+  len = size - start + 1;
+  tsec->ip4_addr_name = kmalloc(len, GFP_KERNEL);
+  if (!tsec->ip4_addr_name)
+   return -ENOMEM;
+  strlcpy(tsec->ip4_addr_name, value+start, len);
+ } else if (strncmp(value, "ip6 ", 4) == 0) {
+  kfree(tsec->ip6_addr_name);
+  start = 4;
+  len = size - start + 1;
+  tsec->ip6_addr_name = kmalloc(len, GFP_KERNEL);
+  if (!tsec->ip6_addr_name)
+   return -ENOMEM;
+  strlcpy(tsec->ip6_addr_name, value+start, len);
+
+ /* the next two are equivalent */
+ } else if (strncmp(value, "slice ", 6) == 0) {
+  val = simple_strtoul(value+6, NULL, 0);
+  tsec->maxtimeslice = val;
+ } else if (strncmp(value, "timeslice ", 10) == 0) {
+  val = simple_strtoul(value+10, NULL, 0);
+  tsec->maxtimeslice = val;
+ } else if (strncmp(value, "nrtask ", 7) == 0) {
+  val = (int) simple_strtol(value+7, NULL, 0);
+  if (val < 1)
+   return -EINVAL;
+  tsec->max_nrtask = val;
+ } else if (strncmp(value, "memlock ", 8) == 0) {
+  val = simple_strtoul(value+8, NULL, 0);
+  tsec->max_memlock = val;
+ } else if (strncmp(value, "data ", 5) == 0) {
+  val = simple_strtoul(value+5, NULL, 0);
+  tsec->max_data = val;
+ } else if (strncmp(value, "nice ", 5) == 0) {
+  val = simple_strtoul(value+5, NULL, 0);
+  tsec->nice = val;
```

```
+ } else if (strncmp(value, "lock", 4) != 0)
+  return -EINVAL;
+
+ return rsize;
+}
+
+static int print_jail_net_info(struct jail_struct *j, char *buf, int maxcnt)
+{
+ int len = 0;
+
+ if (j->ip4_addr_name)
+  len += snprintf(buf, maxcnt, "%s\n", j->ip4_addr_name);
+ if (j->ip6_addr_name)
+  len += snprintf(buf, maxcnt-len, "%s\n", j->ip6_addr_name);
+
+ if (len)
+  return len;
+
+ return snprintf(buf, maxcnt, "No network information\n");
+}
+
+/*
+ * LSM /proc/<pid>/attr read hook.
+ *
+ * /proc/$$/attr/current output:
+ * If the reading process, say process 1001, is in a jail, then
+ *   cat /proc/999/attr/current
+ * will print networking information.
+ * If the reading process, say process 1001, is not in a jail, then
+ *   cat /proc/999/attr/current
+ * will return
+ *   ip:   (ip address of jail)
+ * if 999 is in a jail, or
+ *   -EINVAL
+ * if 999 is not in a jail.
+ *
+ * /proc/$$/attr/exec output:
+ * A process in a jail gets -EINVAL for /proc/$$/attr/exec.
+ * A process not in a jail gets hints on starting a jail.
+ */
+static int
+jail_getprocattr(struct task_struct *p, char *name, void *value, size_t size)
+{
+ struct jail_struct *tsec;
+ int err = 0;
+
+ if (in_jail(current)) {
+  if (strcmp(name, "current") == 0) {
```

```
+   /* provide network info */
+   err = print_jail_net_info(current->security, value,
+    size);
+   return err;
+  }
+  return -EINVAL;  /* let them guess why */
+ }
+
+ if (strcmp(name, "exec") == 0) {
+  /* Print usage some help */
+  err = snprintf(value, size,
+   "Valid keywords:\n"
+   "lock\n"
+   "ip      <ip4-addr>\n"
+   "ip6     <ip6-addr>\n"
+   "nrtask  <max number of tasks in this jail>\n"
+   "nice    <nice level for processes in this jail>\n"
+   "slice   <max timeslice per process in msecs>\n"
+   "data    <max data size per process in bytes>\n"
+   "memlock <max lockable memory per process in bytes>\n");
+  return err;
+ }
+
+ if (strcmp(name, "current"))
+  return -EPERM;
+
+ tsec = p->security;
+ if (!tsec || !(tsec->jail_flags & IN_USE)) {
+  err = snprintf(value, size, "Not Jailed\n");
+ } else {
+  err = snprintf(value, size,
+   "IPv4: %s\nIPv6: %s\n"
+   "max_nrtask %d current nrtask %d max_timeslice %lu "
+   "nice %lu\n"
+   "max_memlock %lu max_data %lu\n",
+   tsec->ip4_addr_name ? tsec->ip4_addr_name : "(none)",
+   tsec->ip6_addr_name ? tsec->ip6_addr_name : "(none)",
+   tsec->max_nrtask, tsec->cur_nrtask, tsec->maxtimeslice,
+   tsec->nice, tsec->max_data, tsec->max_memlock);
+ }
+
+ return err;
+}
+
+/*
+ * Forbid a process in a jail from sending a signal to a process in another
+ * (or no) jail through file sigio.
+ *
```

```
+ * We consider the process which set the fowner to be the one sending the
+ * signal, rather than the one writing to the file.  Therefore we store the
+ * jail of a process during jail_file_set_fowner, then check that against
+ * the jail of the process receiving the signal.
+ */
+static int
+jail_file_send_sigiotask(struct task_struct *tsk, struct fown_struct *fown,
+     int sig)
+{
+ struct file *file;
+
+ if (!in_jail(current))
+  return 0;
+
+ file = container_of(fown, struct file, f_owner);
+ if (file->f_security != tsk->security)
+  return -EPERM;
+
+ return 0;
+}
+
+static int
+jail_file_set_fowner(struct file *file)
+{
+ struct jail_struct *tsec;
+
+ tsec = current->security;
+ file->f_security = tsec;
+ if (tsec)
+  kref_get(&tsec->kref);
+
+ return 0;
+}
+
+static void free_ipc_security(struct kern_ipc_perm *ipc)
+{
+ struct jail_struct *tsec;
+
+ tsec = ipc->security;
+ if (!tsec)
+  return;
+ kref_put(&tsec->kref, release_jail);
+ ipc->security = NULL;
+}
+
+static void free_file_security(struct file *file)
+{
+ struct jail_struct *tsec;
```

```
+
+ tsec = file->f_security;
+ if (!tsec)
+  return;
+ kref_put(&tsec->kref, release_jail);
+ file->f_security = NULL;
+}
+
+static void free_inode_security(struct inode *inode)
+{
+ struct jail_struct *tsec;
+
+ tsec = inode->i_security;
+ if (!tsec)
+  return;
+ kref_put(&tsec->kref, release_jail);
+ inode->i_security = NULL;
+}
+
+/*
+ * LSM ptrace hook:
+ * process in jail may not ptrace process not in the same jail
+ */
+static int
+jail_ptrace (struct task_struct *tracer, struct task_struct *tracee)
+{
+ struct jail_struct *tsec = tracer->security;
+
+ if (tsec && (tsec->jail_flags & IN_USE)) {
+  if (tsec == tracee->security)
+   return 0;
+  return -EPERM;
+ }
+ return 0;
+}
+
+/*
+ * process in jail may only use one (aliased) ip address.  If they try to
+ * attach to 127.0.0.1, that is remapped to their own address.  If some
+ * other address (and not their own), deny permission
+ */
+static int jail_socket_unix_bind(struct socket *sock, struct sockaddr *address,
+  int addrlen);
+
+#define loopbackaddr htonl((127 << 24) | 1)
+
+static inline int jail_inet4_bind(struct socket *sock, struct sockaddr *address,
+ int addrlen, struct jail_struct *tsec)
```

```
+{
+ struct sockaddr_in *inaddr;
+ __u32 sin_addr, jailaddr;
+
+ if (!(tsec->jail_flags & GOT_IPV4))
+  return -EPERM;
+
+ inaddr = (struct sockaddr_in *) address;
+ sin_addr = inaddr->sin_addr.s_addr;
+ jailaddr = tsec->addr4;
+
+ if (sin_addr == jailaddr)
+  return 0;
+
+ if (sin_addr == loopbackaddr || !sin_addr) {
+  bsdj_debug(DBG, "Got a loopback or 0 address\n");
+  sin_addr = jailaddr;
+  bsdj_debug(DBG, "Converted to: %u.%u.%u.%u\n",
+   NIPQUAD(sin_addr));
+  return 0;
+ }
+
+ return -EPERM;
+}
+
+static inline int
+jail_inet6_bind(struct socket *sock, struct sockaddr *address, int addrlen,
+ struct jail_struct *tsec)
+{
+ struct sockaddr_in6 *inaddr6;
+ struct in6_addr *sin6_addr, *jailaddr;
+
+ if (!(tsec->jail_flags & GOT_IPV6))
+  return -EPERM;
+
+ inaddr6 = (struct sockaddr_in6 *) address;
+ sin6_addr = &inaddr6->sin6_addr;
+ jailaddr = &tsec->addr6;
+
+ if (ipv6_addr_cmp(jailaddr, sin6_addr) == 0)
+  return 0;
+
+ if (ipv6_addr_cmp(sin6_addr, &in6addr_loopback) == 0) {
+  ipv6_addr_copy(sin6_addr, jailaddr);
+  return 0;
+ }
+
+ printk(KERN_NOTICE "%s: DENYING\n", __FUNCTION__);
```

```
+ printk(KERN_NOTICE "%s: a %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x "
+  "j %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x\n",
+  __FUNCTION__,
+  NIP6(*sin6_addr),
+  NIP6(*jailaddr));
+
+ return -EPERM;
+}
+
+static int
+jail_socket_bind(struct socket *sock, struct sockaddr *address, int addrlen)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ if (sock->sk->sk_family == AF_UNIX)
+  return jail_socket_unix_bind(sock, address, addrlen);
+
+ if (!(tsec->jail_flags & (GOT_IPV4 | GOT_IPV6)))
+  /* If we want to be strict, we could just
+   * deny net access when lacking a pseudo ip.
+   * For now we just allow it. */
+  return 0;
+
+ switch(address->sa_family) {
+  case AF_INET:
+   return jail_inet4_bind(sock, address, addrlen, tsec);
+
+  case AF_INET6:
+   return jail_inet6_bind(sock, address, addrlen, tsec);
+
+  default:
+   return 0;
+ }
+}
+
+/*
+ * If locked in an ipv6 jail, don't let them use ipv4, and vice versa
+ */
+static int
+jail_socket_create(int family, int type, int protocol, int kern)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || kern || !(tsec->jail_flags & IN_USE) ||
+   !(tsec->jail_flags & (GOT_IPV4 | GOT_IPV6)))
```

```
+  return 0;
+
+ switch(family) {
+  case AF_INET:
+   if (tsec->jail_flags & GOT_IPV4)
+    return 0;
+   return -EPERM;
+  case AF_INET6:
+   if (tsec->jail_flags & GOT_IPV6)
+    return 0;
+   return -EPERM;
+  default:
+   return 0;
+ };
+
+ return 0;
+}
+
+static void
+jail_socket_post_create(struct socket *sock, int family, int type,
+ int protocol, int kern)
+{
+ struct inet_sock *inet;
+ struct ipv6_pinfo *inet6;
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || kern || !(tsec->jail_flags & IN_USE) ||
+   !(tsec->jail_flags & (GOT_IPV4 | GOT_IPV6)))
+  return;
+
+ switch(family) {
+  case AF_INET:
+   inet = inet_sk(sock->sk);
+   inet->saddr = tsec->addr4;
+   break;
+  case AF_INET6:
+   inet6 = inet6_sk(sock->sk);
+   ipv6_addr_copy(&inet6->saddr, &tsec->addr6);
+   break;
+  default:
+   break;
+ };
+
+ return;
+}
+
+static int
+jail_socket_listen(struct socket *sock, int backlog)
```

```
+{
+ struct inet_sock *inet;
+ struct ipv6_pinfo *inet6;
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE) ||
+   !(tsec->jail_flags & (GOT_IPV4 | GOT_IPV6)))
+  return 0;
+
+ switch (sock->sk->sk_family) {
+  case AF_INET:
+   inet = inet_sk(sock->sk);
+   if (inet->saddr == tsec->addr4)
+    return 0;
+   return -EPERM;
+
+  case AF_INET6:
+   inet6 = inet6_sk(sock->sk);
+   if (ipv6_addr_cmp(&inet6->saddr, &tsec->addr6) == 0)
+    return 0;
+   return -EPERM;
+
+  default:
+   return 0;
+
+ }
+}
+
+static void free_sock_security(struct sock *sk)
+{
+ struct jail_struct *tsec;
+
+ tsec = sk->sk_security;
+ if (!tsec)
+  return;
+ kref_put(&tsec->kref, release_jail);
+ sk->sk_security = NULL;
+}
+
+/*
+ * The next three (socket) hooks prevent a process in a jail from sending
+ * data to a abstract unix domain socket which was bound outside the jail.
+ */
+static int
+jail_socket_unix_bind(struct socket *sock, struct sockaddr *address,
+ int addrlen)
+{
+ struct sockaddr_un *sunaddr;
```

```
+ struct jail_struct *tsec;
+
+ if (sock->sk->sk_family != AF_UNIX)
+  return 0;
+
+ sunaddr = (struct sockaddr_un *) address;
+ if (sunaddr->sun_path[0] != 0)
+  return 0;
+
+ tsec = current->security;
+ sock->sk->sk_security = tsec;
+ if (tsec)
+  kref_get(&tsec->kref);
+ return 0;
+}
+
+/*
+ * Note - we deny sends  both from unjailed to jailed, and from jailed
+ * to unjailed.  As well as, of course between different jails.
+ */
+static int
+jail_socket_unix_may_send(struct socket *sock, struct socket *other)
+{
+ struct jail_struct *tsec, *ssec;
+
+ tsec = current->security;  /* jail of sending process */
+ ssec = other->sk->sk_security;  /* jail of receiver */
+
+ if (tsec != ssec)
+  return -EPERM;
+
+ return 0;
+}
+
+static int
+jail_socket_unix_stream_connect(struct socket *sock,
+       struct socket *other, struct sock *newsk)
+{
+ struct jail_struct *tsec, *ssec;
+
+ tsec = current->security;  /* jail of sending process */
+ ssec = other->sk->sk_security;  /* jail of receiver */
+
+ if (tsec != ssec)
+  return -EPERM;
+
+ return 0;
+}
```

```
+
+static int
+jail_mount(char * dev_name, struct nameidata *nd, char * type,
+                        unsigned long flags, void * data)
+{
+ if (in_jail(current))
+  return -EPERM;
+
+ return 0;
+}
+
+static int
+jail_umount(struct vfsmount *mnt, int flags)
+{
+ if (in_jail(current))
+  return -EPERM;
+
+ return 0;
+}
+
+/*
+ * process in jail may not:
+ *   use nice
+ *   change network config
+ *   load/unload modules
+ */
+static int
+jail_capable (struct task_struct *tsk, int cap)
+{
+ if (in_jail(tsk)) {
+  if (cap == CAP_SYS_NICE)
+   return -EPERM;
+  if (cap == CAP_NET_ADMIN)
+   return -EPERM;
+  if (cap == CAP_SYS_MODULE)
+   return -EPERM;
+  if (cap == CAP_SYS_RAWIO)
+   return -EPERM;
+ }
+
+ if (cap_is_fs_cap (cap) ? tsk->fsuid == 0 : tsk->euid == 0)
+  return 0;
+ return -EPERM;
+}
+
+/*
+ * jail_security_task_create:
+ *
```

```
+ * If the current process is ina a jail, and that jail is about to exceed a
+ * maximum number of processes, then refuse to fork.  If the maximum number
+ * of jails is listed as 0, then there is no limit for this jail, and we allow
+ * all forks.
+ */
+static inline int
+jail_security_task_create (unsigned long clone_flags)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ if (tsec->max_nrtask && tsec->cur_nrtask >= tsec->max_nrtask)
+  return -EPERM;
+ return 0;
+}
+
+/*
+ * The child of a process in a jail belongs in the same jail
+ */
+static int
+jail_task_alloc_security(struct task_struct *tsk)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ tsk->security = tsec;
+ kref_get(&tsec->kref);
+ tsec->cur_nrtask++;
+ if (tsec->maxtimeslice) {
+  tsk->signal->rlim[RLIMIT_CPU].rlim_max = tsec->maxtimeslice;
+  tsk->signal->rlim[RLIMIT_CPU].rlim_cur = tsec->maxtimeslice;
+ }
+ if (tsec->max_data) {
+  tsk->signal->rlim[RLIMIT_CPU].rlim_max = tsec->max_data;
+  tsk->signal->rlim[RLIMIT_CPU].rlim_cur = tsec->max_data;
+ }
+ if (tsec->max_memlock) {
+  tsk->signal->rlim[RLIMIT_CPU].rlim_max = tsec->max_memlock;
+  tsk->signal->rlim[RLIMIT_CPU].rlim_cur = tsec->max_memlock;
+ }
+ if (tsec->nice)
+  set_user_nice(current, tsec->nice);
+
+ return 0;
```

```
+}
+
+static int
+jail_bprm_alloc_security(struct linux_binprm *bprm)
+{
+ struct jail_struct *tsec = current->security;
+ int ret;
+
+ if (!tsec)
+  return 0;
+
+ if (tsec->jail_flags & IN_USE)
+  return 0;
+
+ ret = enable_jail(current);
+ if (ret) {
+  /* if we failed, nix out the ip requests */
+  jail_task_free_security(current);
+  return ret;
+ }
+ return 0;
+}
+
+/*
+ * Process in jail may not create devices
+ * Thanks to Brad Spender for pointing out fifos should be allowed.
+ */
+/* TODO: We may want to allow /dev/log, at least... */
+static int
+jail_inode_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
+{
+ if (!in_jail(current))
+  return 0;
+
+ if (S_ISFIFO(mode))
+  return 0;
+
+ return -EPERM;
+}
+
+/* yanked from fs/proc/base.c */
+static unsigned name_to_int(struct dentry *dentry)
+{
+ const char *name = dentry->d_name.name;
+ int len = dentry->d_name.len;
+ unsigned n = 0;
+
+ if (len > 1 && *name == '0')
```

```
+  goto out;
+ while (len-- > 0) {
+  unsigned c = *name++ - '0';
+  if (c > 9)
+   goto out;
+  if (n >= (~0U-9)/10)
+   goto out;
+  n *= 10;
+  n += c;
+ }
+ return n;
+out:
+ return ~0U;
+}
+
+/*
+ * jail_proc_inode_permission:
+ *   called only when current is in a jail, and is trying to reach
+ *   /proc/<pid>.  We check whether <pid> is in the same jail as
+ *   current.  If not, permission is denied.
+ *
+ * NOTE:  On the one hand, the task_to_inode(inode)->i_security
+ * approach seems cleaner, but on the other, this prevents us
+ * from unloading bsdjail for awhile...
+ */
+static int
+jail_proc_inode_permission(struct inode *inode, int mask,
+        struct nameidata *nd)
+{
+ struct jail_struct *tsec = current->security;
+ struct dentry *dentry = nd->dentry;
+ unsigned pid;
+
+ pid = name_to_int(dentry);
+ if (pid == ~0U) {
+  return 0;
+ }
+
+ if (dentry->d_parent != dentry->d_sb->s_root)
+  return 0;
+ if (inode->i_security != tsec)
+  return -ENOENT;
+
+ return 0;
+}
+
+/*
+ * security_task_to_inode:
```

```
+ * Set inode->security = task's jail.
+ */
+static void jail_task_to_inode(struct task_struct *p, struct inode *inode)
+{
+ struct jail_struct *tsec = p->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return;
+ if (inode->i_security)
+  return;
+ kref_get(&tsec->kref);
+ inode->i_security = tsec;
+}
+
+/*
+ * inode_permission:
+ * If we are trying to look into certain /proc files from in a jail, we
+ *  may deny permission.
+ */
+static int
+jail_inode_permission(struct inode *inode, int mask,
+        struct nameidata *nd)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ if (!nd)
+  return 0;
+
+ if (nd->dentry &&
+  strcmp(nd->dentry->d_sb->s_type->name, "proc") == 0) {
+  return jail_proc_inode_permission(inode, mask, nd);
+
+ }
+
+ return 0;
+}
+
+/*
+ * A function which returns -ENOENT if dentry is the dentry for
+ * a /proc/<pid> directory.  It returns 0 otherwise.
+ */
+static inline int
+generic_procpid_check(struct dentry *dentry)
+{
+ struct jail_struct *jail = current->security;
```

```
+ unsigned pid = name_to_int(dentry);
+
+ if (!jail || !(jail->jail_flags & IN_USE))
+  return 0;
+ if (pid == ~0U)
+  return 0;
+ if (strcmp(dentry->d_sb->s_type->name, "proc") != 0)
+  return 0;
+ if (dentry->d_parent != dentry->d_sb->s_root)
+  return 0;
+ if (dentry->d_inode->i_security != jail)
+  return -ENOENT;
+ return 0;
+}
+
+/*
+ * We want getattr to fail on /proc/<pid> to prevent leakage through, for
+ * instance, ls -d.
+ */
+static int
+jail_inode_getattr(struct vfsmount *mnt, struct dentry *dentry)
+{
+ return generic_procpid_check(dentry);
+}
+
+/* This probably is not necessary - /proc does not support xattrs? */
+static int
+jail_inode_getxattr(struct dentry *dentry, char *name)
+{
+ return generic_procpid_check(dentry);
+}
+
+/* process in jail may not send signal to process not in the same jail */
+static int
+jail_task_kill(struct task_struct *p, struct siginfo *info, int sig, u32 secid)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ if (tsec == p->security)
+  return 0;
+
+ if (sig==SIGCHLD)
+  return 0;
+
+ return -EPERM;
```

```
+}
+
+/*
+ * LSM hooks to limit jailed process' abilities to muck with resource
+ * limits
+ */
+static int jail_task_setrlimit (unsigned int resource, struct rlimit *new_rlim)
+{
+ if (!in_jail(current))
+  return 0;
+
+ return -EPERM;
+}
+
+static int jail_task_setscheduler (struct task_struct *p, int policy,
+        struct sched_param *lp)
+{
+ if (!in_jail(current))
+  return 0;
+
+ return -EPERM;
+}
+
+/*
+ * LSM hooks to limit IPC access.
+ */
+
+static inline int
+basic_ipc_security_check(struct kern_ipc_perm *p, struct task_struct *target)
+{
+ struct jail_struct *tsec = target->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+
+ if (p->security != tsec)
+  return -EPERM;
+
+ return 0;
+}
+
+static int
+jail_ipc_permission(struct kern_ipc_perm *ipcp, short flag)
+{
+ return basic_ipc_security_check(ipcp, current);
+}
+
+static int
```

```
+jail_shm_alloc_security (struct shmid_kernel *shp)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+ shp->shm_perm.security = tsec;
+ kref_get(&tsec->kref);
+ return 0;
+}
+
+static void
+jail_shm_free_security (struct shmid_kernel *shp)
+{
+ free_ipc_security(&shp->shm_perm);
+}
+
+static int
+jail_shm_associate (struct shmid_kernel *shp, int shmflg)
+{
+ return basic_ipc_security_check(&shp->shm_perm, current);
+}
+
+static int
+jail_shm_shmctl(struct shmid_kernel *shp, int cmd)
+{
+ if (cmd == IPC_INFO || cmd == SHM_INFO)
+  return 0;
+
+ return basic_ipc_security_check(&shp->shm_perm, current);
+}
+
+static int
+jail_shm_shmat(struct shmid_kernel *shp, char *shmaddr, int shmflg)
+{
+ return basic_ipc_security_check(&shp->shm_perm, current);
+}
+
+static int
+jail_msg_queue_alloc(struct msg_queue *msq)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+ msq->q_perm.security = tsec;
+ kref_get(&tsec->kref);
+ return 0;
```

```
+}
+
+static void
+jail_msg_queue_free(struct msg_queue *msq)
+{
+ free_ipc_security(&msq->q_perm);
+}
+
+static int jail_msg_queue_associate(struct msg_queue *msq, int flag)
+{
+ return basic_ipc_security_check(&msq->q_perm, current);
+}
+
+static int
+jail_msg_queue_msgctl(struct msg_queue *msq, int cmd)
+{
+ if (cmd == IPC_INFO || cmd == MSG_INFO)
+  return 0;
+
+ return basic_ipc_security_check(&msq->q_perm, current);
+}
+
+static int
+jail_msg_queue_msgsnd(struct msg_queue *msq, struct msg_msg *msg, int msqflg)
+{
+ return basic_ipc_security_check(&msq->q_perm, current);
+}
+
+static int
+jail_msg_queue_msgrcv(struct msg_queue *msq, struct msg_msg *msg,
+  struct task_struct *target, long type, int mode)
+
+{
+ return basic_ipc_security_check(&msq->q_perm, target);
+}
+
+static int
+jail_sem_alloc_security(struct sem_array *sma)
+{
+ struct jail_struct *tsec = current->security;
+
+ if (!tsec || !(tsec->jail_flags & IN_USE))
+  return 0;
+ sma->sem_perm.security = tsec;
+ kref_get(&tsec->kref);
+ return 0;
+}
+
```

```
+static void
+jail_sem_free_security(struct sem_array *sma)
+{
+ free_ipc_security(&sma->sem_perm);
+}
+
+static int
+jail_sem_associate(struct sem_array *sma, int semflg)
+{
+ return basic_ipc_security_check(&sma->sem_perm, current);
+}
+
+static int
+jail_sem_semctl(struct sem_array *sma, int cmd)
+{
+ if (cmd == IPC_INFO || cmd == SEM_INFO)
+  return 0;
+ return basic_ipc_security_check(&sma->sem_perm, current);
+}
+
+static int
+jail_sem_semop(struct sem_array *sma, struct sembuf *sops, unsigned nsops,
+ int alter)
+{
+ return basic_ipc_security_check(&sma->sem_perm, current);
+}
+
+static int
+jail_sysctl(struct ctl_table *table, int op)
+{
+ if (!in_jail(current))
+  return 0;
+
+ if (op & 002)
+  return -EPERM;
+
+ return 0;
+}
+
+static struct security_operations bsdjail_security_ops = {
+ .ptrace  =   jail_ptrace,
+ .capable =   jail_capable,
+
+ .task_kill =   jail_task_kill,
+ .task_alloc_security =  jail_task_alloc_security,
+ .task_free_security =  jail_task_free_security,
+ .bprm_alloc_security =  jail_bprm_alloc_security,
+ .task_create =   jail_security_task_create,
```

```
+ .task_to_inode =  jail_task_to_inode,
+
+ .task_setrlimit =  jail_task_setrlimit,
+ .task_setscheduler =  jail_task_setscheduler,
+
+ .setprocattr =                 jail_setprocattr,
+ .getprocattr =                 jail_getprocattr,
+
+ .file_set_fowner =  jail_file_set_fowner,
+ .file_send_sigiotask =  jail_file_send_sigiotask,
+ .file_free_security =  free_file_security,
+
+ .socket_bind =   jail_socket_bind,
+ .socket_listen =  jail_socket_listen,
+ .socket_create =  jail_socket_create,
+ .socket_post_create =  jail_socket_post_create,
+       .unix_stream_connect =  jail_socket_unix_stream_connect,
+ .unix_may_send =  jail_socket_unix_may_send,
+ .sk_free_security =  free_sock_security,
+
+ .inode_mknod =   jail_inode_mknod,
+ .inode_permission =  jail_inode_permission,
+ .inode_free_security =  free_inode_security,
+ .inode_getattr =  jail_inode_getattr,
+ .inode_getxattr =  jail_inode_getxattr,
+ .sb_mount =   jail_mount,
+ .sb_umount =   jail_umount,
+
+ .ipc_permission =  jail_ipc_permission,
+ .shm_alloc_security =   jail_shm_alloc_security,
+ .shm_free_security =   jail_shm_free_security,
+ .shm_associate =  jail_shm_associate,
+ .shm_shmctl =   jail_shm_shmctl,
+ .shm_shmat =   jail_shm_shmat,
+
+ .msg_queue_alloc_security = jail_msg_queue_alloc,
+ .msg_queue_free_security = jail_msg_queue_free,
+ .msg_queue_associate =  jail_msg_queue_associate,
+ .msg_queue_msgctl =  jail_msg_queue_msgctl,
+ .msg_queue_msgsnd =  jail_msg_queue_msgsnd,
+ .msg_queue_msgrcv =  jail_msg_queue_msgrcv,
+
+ .sem_alloc_security =   jail_sem_alloc_security,
+ .sem_free_security =    jail_sem_free_security,
+ .sem_associate =  jail_sem_associate,
+ .sem_semctl =   jail_sem_semctl,
+ .sem_semop =   jail_sem_semop,
+
```

```
+ .sysctl =   jail_sysctl,
+};
+
+static int __init bsdjail_init (void)
+{
+ int rc = 0;
+
+ if (register_security (&bsdjail_security_ops)) {
+  printk (KERN_INFO
+   "Failure registering BSD Jail module with the kernel\n");
+
+  rc = mod_reg_security(MY_NAME, &bsdjail_security_ops);
+  if (rc < 0) {
+   printk (KERN_INFO "Failure registering BSD Jail "
+    " module with primary security module.\n");
+   return -EINVAL;
+  }
+  secondary = 1;
+ }
+ printk (KERN_INFO "BSD Jail module initialized.\n");
+
+ return 0;
+}
+
+static void __exit bsdjail_exit (void)
+{
+ if (secondary) {
+  if (mod_unreg_security (MY_NAME, &bsdjail_security_ops))
+   printk (KERN_INFO "Failure unregistering BSD Jail "
+    " module with primary module.\n");
+ } else {
+  if (unregister_security (&bsdjail_security_ops)) {
+   printk (KERN_INFO "Failure unregistering BSD Jail "
+    "module with the kernel\n");
+  }
+ }
+
+ printk (KERN_INFO "BSD Jail module removed\n");
+}
+
+security_initcall (bsdjail_init);
+module_exit (bsdjail_exit);
+
+MODULE_DESCRIPTION("BSD Jail LSM.");
+MODULE_LICENSE("GPL");
--
1.4.1.1
```