

---

Subject: [PATCH RFC] cgroups: implement device whitelist cgroup+lsn  
Posted by [serue](#) on Thu, 11 Oct 2007 20:04:03 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

Here is an LSM-based alternative to Pavel's device control cgroup, purely for discussion, not for any sort of code review (please :).

thanks,  
-serge

>From 4266131c40b629e3b04c0d9d01569a95fa967e3e Mon Sep 17 00:00:00 2001  
From: Serge E. Hallyn <serue@us.ibm.com>  
Date: Thu, 11 Oct 2007 15:27:48 -0400  
Subject: [PATCH RFC] cgroups: implement device whitelist cgroup+lsn

Implement a cgroup using the LSM interface to enforce open and mknod on device files. Not a line of this code is expected to be used in a final version, this is just a proof of concept.

No stacking is implemented, so to test this you must have

```
CGROUPS=y  
SECURITY=y
```

but all other LSMs =n (no capabilities, no selinux, no rootplug).

This implements a simple device access whitelist. A whitelist entry has 4 fields. 'type' is a (all), c (char), or b (block). 'all' means it applies to all types, all major numbers, and all minor numbers. Major and minor are obvious. Access is a composition of r (read), w (write), and m (mknod).

The root devcgroup starts with rwm to 'all'. A child devcg gets a copy of the parent. Admins can then add and remove devices to the whitelist. Once CAP\_HOST\_ADMIN is introduced it will be needed to add entries as well or remove entries from another cgroup, though just CAP\_SYS\_ADMIN will suffice to remove entries for your own group.

An entry is added by doing "echo <type> <maj> <min> <access>" > devcg.allow, for instance:

```
echo b 7 0 mrw > /cgroups/1/devcg.allow
```

An entry is removed by doing likewise into devcg.deny. Since this is a pure whitelist, not acls, you can only remove entries which exist in the whitelist. You must explicitly

```
echo a 0 0 mrw > /cgroups/1/devcg.deny
```

to remove the "allow all" entry which is automatically inherited from the root cgroup.

While composing this with the ns\_cgroup may seem logical, it may not be the right thing to do. Note that each newly created devcg gets a copy of the parent whitelist. So if you had done

```
mount -t cgroup -o ns,devcg none /cgroups
```

then once a process in /cgroup/1 had done an unshare(CLONE\_NEWNS) it would be under /cgroup/1/node\_<pid> if an admin did

```
echo b 7 0 m > /cgroups/1/devcg.deny
```

then the entry would still be in the whitelist for /cgroups/1/node\_<pid>. Something to discuss if we get that far before nixing this whole idea.

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>

---

```
include/linux/cgroup_subsys.h | 6 +
init/Kconfig                  | 7 +
kernel/Makefile               | 1 +
kernel/dev_cgroup.c           | 554 +++++
4 files changed, 568 insertions(+), 0 deletions(-)
create mode 100644 kernel/dev_cgroup.c
```

```
diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
```

```
index d822977..cf55cb2 100644
```

```
--- a/include/linux/cgroup_subsys.h
```

```
+++ b/include/linux/cgroup_subsys.h
```

```
@ @ -36,3 +36,9 @ @ SUBSYS(mem_cgroup)
#endif
```

```
/* */
```

```
+
```

```
+#ifdef CONFIG_CGROUP_DEV
```

```
+SUBSYS(devcg)
```

```
+#endif
```

```
+
```

```
/* */
```

```
diff --git a/init/Kconfig b/init/Kconfig
```

```
index 6bb603a..0b3b684 100644
```

```
--- a/init/Kconfig
```

```
+++ b/init/Kconfig
```

```
@ @ -319,6 +319,13 @ @ config CPUSETS
```

Say N if unsure.

```
+config CGROUP_DEV
+ bool "Device controller for cgroups"
+ depends on CGROUPS && SECURITY && EXPERIMENTAL
+ help
+   Provides a cgroup implementing whitelists for devices which
+   a process in the cgroup can mknod or open.
+
config FAIR_GROUP_SCHED
  bool "Fair group CPU scheduler"
  default y
diff --git a/kernel/Makefile b/kernel/Makefile
index 76f782f..6ded46d 100644
--- a/kernel/Makefile
+++ b/kernel/Makefile
@@ -43,6 +43,7 @@ obj-$(CONFIG_CGROUP_DEBUG) += cgroup_debug.o
obj-$(CONFIG_CPUSETS) += cpuset.o
obj-$(CONFIG_CGROUP_CPUACCT) += cpu_acct.o
obj-$(CONFIG_CGROUP_NS) += ns_cgroup.o
+obj-$(CONFIG_CGROUP_DEV) += dev_cgroup.o
obj-$(CONFIG_IKCONFIG) += configs.o
obj-$(CONFIG_STOP_MACHINE) += stop_machine.o
obj-$(CONFIG_AUDIT) += audit.o auditfilter.o
diff --git a/kernel/dev_cgroup.c b/kernel/dev_cgroup.c
new file mode 100644
index 0000000..87c8fb4
--- /dev/null
+++ b/kernel/dev_cgroup.c
@@ -0,0 +1,554 @@
+/*
+ * dev_cgroup.c - device cgroup subsystem
+ *
+ * Copyright 2007 IBM Corp
+ */
+
+#include <linux/module.h>
+#include <linux/cgroup.h>
+#include <linux/fs.h>
+#include <linux/list.h>
+#include <linux/security.h>
+
+#include <asm/uaccess.h>
+
+#define ACC_MKNOD 1
+#define ACC_READ 2
+#define ACC_WRITE 4
```

```

+
+ #define DEV_BLOCK 1
+ #define DEV_CHAR 2
+ #define DEV_ALL 4 /* this represents all devices */
+
+ /*
+  * whitelist locking rules:
+  * cgroup_lock() cannot be taken under cgroup->lock.
+  * cgroup->lock can be taken with or without cgroup_lock().
+  *
+  * modifications always require cgroup_lock
+  * modifications to a list which is visible require the
+  * cgroup->lock *and* cgroup_lock()
+  * walking the list requires cgroup->lock or cgroup_lock().
+  *
+  * reasoning: dev_whitelist_copy() needs to kcalloc, so needs
+  * a mutex, which the cgroup_lock() is. Since modifying
+  * a visible list requires both locks, either lock can be
+  * taken for walking the list. Since the wh->spinlock is taken
+  * for modifying a public-accessible list, the spinlock is
+  * sufficient for just walking the list.
+  */
+
+ struct dev_whitelist_item {
+ u32 major, minor;
+ short type;
+ short access;
+ struct list_head list;
+ };
+
+ struct dev_cgroup {
+ struct cgroup_subsys_state css;
+ struct list_head whitelist;
+ spinlock_t lock;
+ };
+
+ struct cgroup_subsys devcg_subsys;
+
+ static inline struct dev_cgroup *cgroup_to_devcg(
+ struct cgroup *cgroup)
+ {
+ return container_of(cgroup_subsys_state(cgroup, devcg_subsys_id),
+ struct dev_cgroup, css);
+ }
+
+ /*
+  * Once 64-bit caps and CAP_HOST_ADMIN exist, we will be
+  * requiring (CAP_HOST_ADMIN|CAP_MKNOD) to create a device

```

```

+ * not in the whitelist, * (CAP_HOST_ADMIN|CAP_SYS_ADMIN)
+ * to edit the whitelist,
+ */
+static int devcg_can_attach(struct cgroup_subsys *ss,
+ struct cgroup *new_cgroup, struct task_struct *task)
+{
+ struct cgroup *orig;
+
+ if (current != task) {
+ if (!cgroup_is_descendant(new_cgroup))
+ return -EPERM;
+ }
+
+ if (atomic_read(&new_cgroup->count) != 0)
+ return -EPERM;
+
+ orig = task_cgroup(task, devcg_subsys_id);
+ if (orig && orig != new_cgroup->parent)
+ return -EPERM;
+
+ return 0;
+}
+
+/*
+ * called under cgroup_lock()
+ */
+int dev_whitelist_copy(struct list_head *dest, struct list_head *orig)
+{
+ struct dev_whitelist_item *wh, *tmp, *new;
+
+ list_for_each_entry(wh, orig, list) {
+ new = kmalloc(sizeof(*wh), GFP_KERNEL);
+ if (!new)
+ goto free_and_exit;
+ new->major = wh->major;
+ new->minor = wh->minor;
+ new->type = wh->type;
+ new->access = wh->access;
+ list_add_tail(&new->list, dest);
+ }
+
+ return 0;
+
+free_and_exit:
+ list_for_each_entry_safe(wh, tmp, dest, list) {
+ list_del(&wh->list);
+ kfree(wh);
+ }
+}

```

```

+ return -ENOMEM;
+}
+
+/* Stupid prototype - don't bother combining existing entries */
+/*
+ * called under cgroup_lock()
+ * since the list is visible to other tasks, we need the spinlock also
+ */
+void dev_whitelist_add(struct dev_cgroup *dev_cgroup, struct dev_whitelist_item *wh)
+{
+ spin_lock(&dev_cgroup->lock);
+ list_add_tail(&wh->list, &dev_cgroup->whitelist);
+ spin_unlock(&dev_cgroup->lock);
+}
+
+/*
+ * called under cgroup_lock()
+ * since the list is visible to other tasks, we need the spinlock also
+ */
+void dev_whitelist_rm(struct dev_cgroup *dev_cgroup, struct dev_whitelist_item *wh)
+{
+ struct dev_whitelist_item *walk, *tmp;
+
+ spin_lock(&dev_cgroup->lock);
+ list_for_each_entry_safe(walk, tmp, &dev_cgroup->whitelist, list) {
+ if (walk->type & DEV_ALL) {
+ list_del(&walk->list);
+ kfree(walk);
+ continue;
+ }
+ if (walk->type != wh->type)
+ continue;
+ if (walk->major != wh->major || walk->minor != wh->minor)
+ continue;
+ walk->access &= ~wh->access;
+ if (!walk->access) {
+ list_del(&walk->list);
+ kfree(walk);
+ }
+ }
+ spin_unlock(&dev_cgroup->lock);
+}
+
+/*
+ * Rules: you can only create a cgroup if
+ * 1. you are capable(CAP_SYS_ADMIN)
+ * 2. the target cgroup is a descendant of your own cgroup
+ */

```

```

+ * Note: called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static struct cgroup_subsys_state *devcg_create(struct cgroup_subsys *ss,
+ struct cgroup *cgroup)
+{
+ struct dev_cgroup *dev_cgroup, *parent_dev_cgroup;
+ struct cgroup *parent_cgroup;
+ int ret;
+
+ if (!capable(CAP_SYS_ADMIN))
+ return ERR_PTR(-EPERM);
+ if (!cgroup_is_descendant(cgroup))
+ return ERR_PTR(-EPERM);
+
+ dev_cgroup = kzalloc(sizeof(*dev_cgroup), GFP_KERNEL);
+ if (!dev_cgroup)
+ return ERR_PTR(-ENOMEM);
+ INIT_LIST_HEAD(&dev_cgroup->whitelist);
+ parent_cgroup = cgroup->parent;
+
+ if (parent_cgroup == NULL) {
+ struct dev_whitelist_item *wh;
+ wh = kmalloc(sizeof(*wh), GFP_KERNEL);
+ wh->minor = wh->major = 0;
+ wh->type = DEV_ALL;
+ wh->access = ACC_MKNOD | ACC_READ | ACC_WRITE;
+ list_add(&wh->list, &dev_cgroup->whitelist);
+ } else {
+ parent_dev_cgroup = cgroup_to_devcg(parent_cgroup);
+ ret = dev_whitelist_copy(&dev_cgroup->whitelist,
+ &parent_dev_cgroup->whitelist);
+ if (ret) {
+ kfree(dev_cgroup);
+ return ERR_PTR(ret);
+ }
+ }
+
+ spin_lock_init(&dev_cgroup->lock);
+ return &dev_cgroup->css;
+}
+
+static void devcg_destroy(struct cgroup_subsys *ss,
+ struct cgroup *cgroup)
+{
+ struct dev_cgroup *dev_cgroup;
+ struct dev_whitelist_item *wh, *tmp;
+
+ dev_cgroup = cgroup_to_devcg(cgroup);

```

```

+ list_for_each_entry_safe(wh, tmp, &dev_cgroup->whitelist, list) {
+ list_del(&wh->list);
+ kfree(wh);
+ }
+ kfree(dev_cgroup);
+}
+
+#define DEVCG_ALLOW 1
+#define DEVCG_DENY 2
+
+void set_access(char *acc, short access)
+{
+ int idx = 0;
+ memset(acc, 0, 4);
+ if (access & ACC_READ)
+ acc[idx++] = 'r';
+ if (access & ACC_WRITE)
+ acc[idx++] = 'w';
+ if (access & ACC_MKNOD)
+ acc[idx++] = 'm';
+}
+
+char type_to_char(short type)
+{
+ if (type == DEV_ALL)
+ return 'a';
+ if (type == DEV_CHAR)
+ return 'c';
+ if (type == DEV_BLOCK)
+ return 'b';
+ return 'X';
+}
+
+char *print_whitelist(struct dev_cgroup *devcgroup, int *len)
+{
+ char *buf, *s, acc[4];
+ struct dev_whitelist_item *wh;
+ int ret;
+ int count = 0;
+
+ buf = kmalloc(4096, GFP_KERNEL);
+ if (!buf)
+ return ERR_PTR(-ENOMEM);
+ s = buf;
+ *s = '\0';
+ *len = 0;
+
+ spin_lock(&devcgroup->lock);

```



```

+ list_for_each_entry(wh, &devcgroup->whitelist, list) {
+ set_access(acc, wh->access);
+ printk(KERN_NOTICE "%s (count%d): whtype %hd maj %u min %u acc %hd\n",
__FUNCTION__,
+ count, wh->type, wh->major, wh->minor, wh->access);
+ ret = snprintf(s, 4095-(s-buf), "%c %u %u %s\n",
+ type_to_char(wh->type), wh->major, wh->minor, acc);
+ if (s+ret >= buf+4095) {
+ kfree(buf);
+ buf = ERR_PTR(-ENOMEM);
+ break;
+ }
+ s += ret;
+ *len += ret;
+ count++;
+ }
+ spin_unlock(&devcgroup->lock);
+
+ return buf;
+}
+
+static ssize_t devcg_access_read(struct cgroup *cgroup,
+ struct cftype *cft, struct file *file,
+ char __user *userbuf, size_t nbytes, loff_t *ppos)
+{
+ struct dev_cgroup *devcgrp = cgroup_to_devcg(cgroup);
+ int filetype = cft->private;
+ char *buffer;
+ int len, retval;
+
+ if (filetype != DEVCG_ALLOW)
+ return -EINVAL;
+ buffer = print_whitelist(devcgrp, &len);
+ if (IS_ERR(buffer))
+ return PTR_ERR(buffer);
+
+ retval = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, len);
+ kfree(buffer);
+ return retval;
+}
+
+static inline short convert_access(char *acc)
+{
+ short access = 0;
+
+ while (*acc) {
+ switch (*acc) {
+ case 'r':

```

```

+ case 'R': access |= ACC_READ; break;
+ case 'w':
+ case 'W': access |= ACC_WRITE; break;
+ case 'm':
+ case 'M': access |= ACC_MKNOD; break;
+ case '\n': break;
+ default:
+ return -EINVAL;
+ }
+ acc++;
+ }
+
+ return access;
+}
+
+static inline short convert_type(char intype)
+{
+ short type = 0;
+ switch(intype) {
+ case 'a': type = DEV_ALL; break;
+ case 'c': type = DEV_CHAR; break;
+ case 'b': type = DEV_BLOCK; break;
+ default: type = -EACCES; break;
+ }
+ return type;
+}
+
+/*
+ * Current rules:
+ * CAP_SYS_ADMIN needed for all writes.
+ * when we have CAP_HOST_ADMIN, the rules will become:
+ * if (!writetoself)
+ *     require capable(CAP_HOST_ADMIN | CAP_SYS_ADMIN);
+ * if (is_allow)
+ *     require capable(CAP_HOST_ADMIN | CAP_SYS_ADMIN);
+ * require capable(CAP_SYS_ADMIN);
+ */
+static int have_write_permission(int is_allow, int writetoself)
+{
+ if (!capable(CAP_SYS_ADMIN))
+ return 0;
+ return 1;
+}
+
+static ssize_t devcg_access_write(struct cgroup *cgroup, struct cftype *cft,
+ struct file *file, const char __user *userbuf,
+ size_t nbytes, loff_t *ppos)
+{

```

```

+ struct cgroup *cur_cgroup;
+ struct dev_cgroup *devcgrp, *cur_devcgroup;
+ int filetype = cft->private;
+ char *buffer, acc[4];
+ int retval = 0;
+ int nitems;
+ char type;
+ struct dev_whitelist_item *wh;
+
+ devcgrp = cgroup_to_devcg(cgroup);
+ cur_cgroup = task_cgroup(current, devcg_subsys.subsys_id);
+ cur_devcgroup = cgroup_to_devcg(cur_cgroup);
+
+ if (!have_write_permission(filetype==DEVCG_ALLOW, cur_devcgroup==devcgrp))
+ return -EPERM;
+
+ buffer = kmalloc(nbytes+1, GFP_KERNEL);
+ if (!buffer)
+ return -ENOMEM;
+
+ wh = kmalloc(sizeof(*wh), GFP_KERNEL);
+ if (!wh) {
+ kfree(buffer);
+ return -ENOMEM;
+ }
+
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+ retval = -EFAULT;
+ goto out1;
+ }
+ buffer[nbytes] = 0; /* nul-terminate */
+
+ cgroup_lock();
+ if (cgroup_is_removed(cgroup)) {
+ retval = -ENODEV;
+ goto out2;
+ }
+
+ memset(wh, 0, sizeof(*wh));
+ memset(acc, 0, 4);
+ nitems = sscanf(buffer, "%c %u %u %3s", &type, &wh->major, &wh->minor, acc);
+ retval = -EINVAL;
+ if (nitems != 4)
+ goto out2;
+ wh->type = convert_type(type);
+ if (wh->type < 0)
+ goto out2;
+ wh->access = convert_access(acc);

```

```

+ if (wh->access < 0)
+ goto out2;
+ retval = 0;
+ switch (filetype) {
+ case DEVCG_ALLOW:
+ printk(KERN_NOTICE "%s: add whtype %hd maj %u min %u acc %hd\n", __FUNCTION__,
+ wh->type, wh->major, wh->minor, wh->access);
+ dev_whitelist_add(devcgrp, wh);
+ break;
+ case DEVCG_DENY:
+ dev_whitelist_rm(devcgrp, wh);
+ break;
+ default:
+ retval = -EINVAL;
+ goto out2;
+ }
+
+ if (retval == 0)
+ retval = nbytes;
+
+out2:
+ cgroup_unlock();
+out1:
+ kfree(buffer);
+ return retval;
+}
+
+static struct cftype dev_cgroup_files[] = {
+ {
+ .name = "allow",
+ .read = devcg_access_read,
+ .write = devcg_access_write,
+ .private = DEVCG_ALLOW,
+ },
+ {
+ .name = "deny",
+ .write = devcg_access_write,
+ .private = DEVCG_DENY,
+ },
+ };
+
+static int devcg_populate(struct cgroup_subsys *ss,
+ struct cgroup *cont)
+{
+ return cgroup_add_files(cont, ss, dev_cgroup_files,
+ ARRAY_SIZE(dev_cgroup_files));
+}
+

```

```

+struct cgroup_subsys devcg_subsys = {
+ .name = "devcg",
+ .can_attach = devcg_can_attach,
+ .create = devcg_create,
+ .destroy = devcg_destroy,
+ .populate = devcg_populate,
+ .subsys_id = devcg_subsys_id,
+};
+
+static int devcgroup_inode_permission(struct inode *inode, int mask,
+      struct nameidata *nd)
+{
+ struct cgroup *cgroup;
+ struct dev_cgroup *dev_cgroup;
+ struct dev_whitelist_item *wh;
+
+ dev_t device = inode->i_rdev;
+ if (!device)
+ return 0;
+ if (!S_ISBLK(inode->i_mode) && !S_ISCHR(inode->i_mode))
+ return 0;
+ cgroup = task_cgroup(current, devcg_subsys.subsys_id);
+ dev_cgroup = cgroup_to_devcg(cgroup);
+ if (!dev_cgroup)
+ return 0;
+
+ spin_lock(&dev_cgroup->lock);
+ /* if capable(CAP_HOST_ADMIN) return 0; */
+ list_for_each_entry(wh, &dev_cgroup->whitelist, list) {
+ if (wh->type & DEV_ALL)
+ goto acc_check;
+ if (S_ISBLK(inode->i_mode) && !(wh->type & DEV_BLOCK))
+ continue;
+ if (S_ISCHR(inode->i_mode) && !(wh->type & DEV_CHAR))
+ continue;
+ if (wh->major != imajor(inode) || wh->minor != iminor(inode))
+ continue;
+acc_check:
+ if ((mask & MAY_WRITE) && !(wh->access & ACC_WRITE))
+ continue;
+ if ((mask & MAY_READ) && !(wh->access & ACC_READ))
+ continue;
+ spin_unlock(&dev_cgroup->lock);
+ return 0;
+ }
+ spin_unlock(&dev_cgroup->lock);
+
+ printk(KERN_NOTICE "%s: %d denied %d access to %s (%lu)\n", __FUNCTION__,

```

```

+ current->pid, mask, nd ? nd->dentry->d_name.name : "null",
+ inode->i_ino);
+ return -EPERM;
+}
+
+static int devcgroup_inode_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
+{
+ struct cgroup *cgroup;
+ struct dev_cgroup *dev_cgroup;
+ struct dev_whitelist_item *wh;
+
+ /* if capable(CAP_HOST_ADMIN) return 0; */
+ cgroup = task_cgroup(current, devcg_subsys.subsys_id);
+ dev_cgroup = cgroup_to_devcg(cgroup);
+ if (!dev_cgroup)
+ return 0;
+
+ spin_lock(&dev_cgroup->lock);
+ list_for_each_entry(wh, &dev_cgroup->whitelist, list) {
+ if (wh->type & DEV_ALL)
+ goto ok;
+ if (S_ISBLK(mode) && !(wh->type & DEV_BLOCK))
+ continue;
+ if (S_ISCHR(mode) && !(wh->type & DEV_CHAR))
+ continue;
+ if (wh->major != MAJOR(dev) || wh->minor != MINOR(dev))
+ continue;
+ if (wh->access & ACC_MKNOD)
+ goto ok;
+ }
+ spin_unlock(&dev_cgroup->lock);
+
+ printk(KERN_NOTICE "%s: %d denied %d access to (%d %d)\n", __FUNCTION__,
+ current->pid, mode, MAJOR(dev), MINOR(dev));
+ return -EPERM;
+
+ok:
+ spin_unlock(&dev_cgroup->lock);
+ return 0;
+}
+
+static struct security_operations devcgroup_security_ops = {
+ .inode_mknod = devcgroup_inode_mknod,
+ .inode_permission = devcgroup_inode_permission,
+};
+
+static int __init dev_cgroup_security_init (void)
+{

```

```
+ /* register ourselves with the security framework */
+ if (register_security (&devcgroup_security_ops)) {
+   printk (KERN_INFO "Failure registering device cgroup lsm\n");
+   return -EINVAL;
+ }
+ printk (KERN_INFO "Device cgroup LSM initialized\n");
+ return 0;
+}
+
+security_initcall (dev_cgroup_security_init);
--
1.5.1.1.GIT
```

---

Containers mailing list  
Containers@lists.linux-foundation.org  
<https://lists.linux-foundation.org/mailman/listinfo/containers>

---