Subject: [PATCH] cgroups: implement device whitelist (v4)
Posted by serue on Mon, 17 Mar 2008 18:07:22 GMT
View Forum Message <> Reply to Message

Implement a cgroup to track and enforce open and mknod restrictions on device
files. A device cgroup associates a device access whitelist with each
cgroup. A whitelist entry has 4 fields. 'type' is a (all), c (char), or
b (block). 'all' means it applies to all types and all major and minor
numbers. Major and minor are either an integer or * for all.
Access is a composition of r (read), w (write), and m (mknod).

The root device cgroup starts with rwm to 'all'. A child devcg gets
a copy of the parent. Admins can then remove devices from the
whitelist or add new entries. A child cgroup can never receive a
device access which is denied its parent. However when a device
access is removed from a parent it will not also be removed from the
child(ren).

An entry is added using devices.allow, and removed using
devices.deny. For instance

echo 'c 1:3 mr' > /cgroups/1/devices.allow

allows cgroup 1 to read and mknod the device usually known as
/dev/null. Doing

echo a > /cgroups/1/devices.deny

will remove the default 'a *:* mrw' entry.

CAP_SYS_ADMIN is needed to change permissions or move another task
to a new cgroup. A cgroup may not be granted more permissions than
the cgroup's parent has. Any task can move itself between cgroups.
This won't be sufficient, but we can decide the best way to
adequately restrict movement later.

The parsing of devices.allow/deny needs to be cleaned up a bit and
Documented. I'd like to get an idea whether this approach is otherwise
acceptable.

Changelog:
Mar 17 2008: Place specific device cgroup hooks next to
security_inode_{mknod,permission} rather than using
the security hooks.
Also remove most of the controls over tasks moving
between cgroups and playing with the allow and deny
permissions.
Switch to major:minor format.

```
  Rename devcg to 'devices' to conform to cgroup naming.
 Mar 13 2008: move the dev_cgroup support into
  capability hooks instead of having it
  as a separate security module.
  Support root_plug with devcg.
  Note that due to this change, devcg will
  not be enforcing if the dummy module is
  loaded, or if selinux is loaded without
  capabilities.
 Mar 12 2008: allow dev_cgroup lsm to be used when
  SECURITY=n, and allow stacking with SELinux
  and Smack.  Don't work too hard in Kconfig
  to prevent a warning when smack+devcg are
  both compiled in, worry about that later.

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
---
 fs/namei.c                  |    9 +
 include/linux/cgroup_subsys.h |    6 +
 include/linux/device_cgroup.h |   12 +
 init/Kconfig                |    7 +
 security/Makefile           |    1 +
 security/device_cgroup.c    |  597 ++++++++++++++++++++++++++++++++++++++++++
 6 files changed, 632 insertions(+), 0 deletions(-)
 create mode 100644 include/linux/device_cgroup.h
 create mode 100644 security/device_cgroup.c

diff --git a/fs/namei.c b/fs/namei.c
index dfb3cb8..6caed32 100644
--- a/fs/namei.c
+++ b/fs/namei.c
@@ -30,6 +30,7 @@
 #include <linux/capability.h>
 #include <linux/file.h>
 #include <linux/fcntl.h>
+#include <linux/device_cgroup.h>
 #include <asm/namei.h>
 #include <asm/uaccess.h>

@@ -281,6 +282,10 @@ int permission(struct inode *inode, int mask, struct nameidata *nd)
  if (retval)
   return retval;

+ retval = devcgroup_inode_permission(inode, mask);
+ if (retval)
+  return retval;
+
  return security_inode_permission(inode, mask, nd);
```

```
 }

@@ -2028,6 +2033,10 @@ int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t
dev)
  if (!dir->i_op || !dir->i_op->mknod)
   return -EPERM;

+ error = devcgroup_inode_mknod(mode, dev);
+ if (error)
+  return error;
+
  error = security_inode_mknod(dir, dentry, mode, dev);
  if (error)
   return error;
diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
index 1ddebfc..e287745 100644
--- a/include/linux/cgroup_subsys.h
+++ b/include/linux/cgroup_subsys.h
@@ -42,3 +42,9 @@ SUBSYS(mem_cgroup)
 #endif

 /* */
+
+#ifdef CONFIG_CGROUP_DEVICE
+SUBSYS(devices)
+#endif
+
+/* */
diff --git a/include/linux/device_cgroup.h b/include/linux/device_cgroup.h
new file mode 100644
index 0000000..0b0d9c3
--- /dev/null
+++ b/include/linux/device_cgroup.h
@@ -0,0 +1,12 @@
+#include <linux/module.h>
+#include <linux/fs.h>
+
+#ifdef CONFIG_CGROUP_DEVICE
+extern int devcgroup_inode_permission(struct inode *inode, int mask);
+extern int devcgroup_inode_mknod(int mode, dev_t dev);
+#else
+static inline int devcgroup_inode_permission(struct inode *inode, int mask)
+{ return 0; }
+static inline int devcgroup_inode_mknod(int mode, dev_t dev)
+{ return 0; }
+#endif
diff --git a/init/Kconfig b/init/Kconfig
index 009f2d8..30868cd 100644
```

```
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -298,6 +298,13 @@ config CGROUP_NS
          for instance virtual servers and checkpoint/restart
          jobs.

+config CGROUP_DEVICE
+ bool "Device controller for cgroups"
+ depends on CGROUPS && EXPERIMENTAL
+ help
+   Provides a cgroup implementing whitelists for devices which
+   a process in the cgroup can mknod or open.
+
 config CPUSETS
  bool "Cpuset support"
  depends on SMP && CGROUPS
diff --git a/security/Makefile b/security/Makefile
index 9e8b025..7ef1107 100644
--- a/security/Makefile
+++ b/security/Makefile
@@ -18,3 +18,4 @@ obj-$(CONFIG_SECURITY_SELINUX)  += selinux/built-in.o
 obj-$(CONFIG_SECURITY_SMACK)  += commoncap.o smack/built-in.o
 obj-$(CONFIG_SECURITY_CAPABILITIES) += commoncap.o capability.o
 obj-$(CONFIG_SECURITY_ROOTPLUG)  += commoncap.o root_plug.o
+obj-$(CONFIG_CGROUP_DEVICE)  += device_cgroup.o
diff --git a/security/device_cgroup.c b/security/device_cgroup.c
new file mode 100644
index 0000000..33d8fd8
--- /dev/null
+++ b/security/device_cgroup.c
@@ -0,0 +1,597 @@
+/*
+ * dev_cgroup.c - device cgroup subsystem
+ *
+ * Copyright 2007 IBM Corp
+ */
+
+#include <linux/device_cgroup.h>
+#include <linux/cgroup.h>
+#include <linux/ctype.h>
+#include <linux/list.h>
+#include <asm/uaccess.h>
+
+#define ACC_MKNOD 1
+#define ACC_READ  2
+#define ACC_WRITE 4
+#define ACC_MASK (ACC_MKNOD | ACC_READ | ACC_WRITE)
+
```

```
+#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 kmalloc, 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;
+};
+
+static inline struct dev_cgroup *cgroup_to_devcgroup(
+  struct cgroup *cgroup)
+{
+ return container_of(cgroup_subsys_state(cgroup, devices_subsys_id),
+       struct dev_cgroup, css);
+}
+
+
+struct cgroup_subsys devices_subsys;
+
+static int devcgroup_can_attach(struct cgroup_subsys *ss,
+  struct cgroup *new_cgroup, struct task_struct *task)
+{
```

```
+
+ if (current != task && !capable(CAP_SYS_ADMIN))
+   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
+ */
+int dev_whitelist_add(struct dev_cgroup *dev_cgroup,
+   struct dev_whitelist_item *wh)
+{
+ struct dev_whitelist_item *whcopy;
+
+ whcopy = kmalloc(sizeof(*whcopy), GFP_KERNEL);
+ if (!whcopy)
+  return -ENOMEM;
```

```
+
+ memcpy(whcopy, wh, sizeof(*whcopy));
+ spin_lock(&dev_cgroup->lock);
+ list_add_tail(&whcopy->list, &dev_cgroup->whitelist);
+ spin_unlock(&dev_cgroup->lock);
+ return 0;
+}
+
+/*
+ * 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)
+   goto remove;
+  if (walk->type != wh->type)
+   continue;
+  if (walk->major != ~0 && walk->major != wh->major)
+   continue;
+  if (walk->minor != ~0 && walk->minor != wh->minor)
+   continue;
+
+remove:
+  walk->access &= ~wh->access;
+  if (!walk->access) {
+   list_del(&walk->list);
+   kfree(walk);
+  }
+ }
+ spin_unlock(&dev_cgroup->lock);
+}
+
+/*
+ * called from kernel/cgroup.c with cgroup_lock() held.
+ */
+static struct cgroup_subsys_state *devcgroup_create(struct cgroup_subsys *ss,
+      struct cgroup *cgroup)
+{
+ struct dev_cgroup *dev_cgroup, *parent_dev_cgroup;
+ struct cgroup *parent_cgroup;
+ int ret;
+
```

```
+ 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_devcgroup(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 devcgroup_destroy(struct cgroup_subsys *ss,
+   struct cgroup *cgroup)
+{
+ struct dev_cgroup *dev_cgroup;
+ struct dev_whitelist_item *wh, *tmp;
+
+ dev_cgroup = cgroup_to_devcgroup(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';
+}
+
+static void set_majmin(char *str, int len, unsigned m)
+{
+ memset(str, 0, len);
+ if (m == ~0)
+  sprintf(str, "*");
+ else
+  snprintf(str, len, "%d", m);
+}
+
+char *print_whitelist(struct dev_cgroup *devcgroup, int *len)
+{
+ char *buf, *s, acc[4];
+ struct dev_whitelist_item *wh;
+ int ret;
+ int count = 0;
+ char maj[10], min[10];
+
+ 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);
+  set_majmin(maj, 10, wh->major);
+  set_majmin(min, 10, wh->minor);
```

```
+  ret = snprintf(s, 4095-(s-buf), "%c %s:%s %s\n",
+   type_to_char(wh->type), maj, min, 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 devcgroup_access_read(struct cgroup *cgroup,
+   struct cftype *cft, struct file *file,
+   char __user *userbuf, size_t nbytes, loff_t *ppos)
+{
+ struct dev_cgroup *devcgroup = cgroup_to_devcgroup(cgroup);
+ int filetype = cft->private;
+ char *buffer;
+ int len, retval;
+
+ if (filetype != DEVCG_ALLOW)
+  return -EINVAL;
+ buffer = print_whitelist(devcgroup, &len);
+ if (IS_ERR(buffer))
+  return PTR_ERR(buffer);
+
+ retval = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, len);
+ kfree(buffer);
+ return retval;
+}
+
+/*
+ * may_access_whitelist:
+ * does the access granted to dev_cgroup c contain the access
+ * requested in whitelist item refwh.
+ * return 1 if yes, 0 if no.
+ * call with c->lock held
+ */
+static int may_access_whitelist(struct dev_cgroup *c,
+          struct dev_whitelist_item *refwh)
+{
+ struct dev_whitelist_item *whitem;
+
```

```
+ list_for_each_entry(whitem, &c->whitelist, list) {
+  if (whitem->type & DEV_ALL)
+   return 1;
+  if ((refwh->type & DEV_BLOCK) && !(whitem->type & DEV_BLOCK))
+   continue;
+  if ((refwh->type & DEV_CHAR) && !(whitem->type & DEV_CHAR))
+   continue;
+  if (whitem->major != ~0 && whitem->major != refwh->major)
+   continue;
+  if (whitem->minor != ~0 && whitem->minor != refwh->minor)
+   continue;
+  if (refwh->access & (~(whitem->access | ACC_MASK)))
+   continue;
+  return 1;
+ }
+ return 0;
+}
+
+/*
+ * parent_has_perm:
+ * when adding a new allow rule to a device whitelist, the rule
+ * must be allowed in the parent device
+ */
+static int parent_has_perm(struct cgroup *childcg,
+       struct dev_whitelist_item *wh)
+{
+ struct cgroup *pcg = childcg->parent;
+ struct dev_cgroup *parent;
+ int ret;
+
+ if (!pcg)
+  return 1;
+ parent = cgroup_to_devcgroup(pcg);
+ spin_lock(&parent->lock);
+ ret = may_access_whitelist(parent, wh);
+ spin_unlock(&parent->lock);
+ return ret;
+}
+
+/*
+ * Modify the whitelist using allow/deny rules.
+ * CAP_SYS_ADMIN is needed for this.  It's at least separate from CAP_MKNOD
+ * so we can give a container CAP_MKNOD to let it create devices but not
+ * modify the whitelist.
+ * It seems likely we'll want to add a CAP_CONTAINER capability to allow
+ * us to also grant CAP_SYS_ADMIN to containers without giving away the
+ * device whitelist controls, but for now we'll stick with CAP_SYS_ADMIN
+ *
```

```
+ * Taking rules away is always allowed (given CAP_SYS_ADMIN).  Granting
+ * new access is only allowed if you're in the top-level cgroup, or your
+ * parent cgroup has the access you're asking for.
+ */
+static ssize_t devcgroup_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 *devcgroup, *cur_devcgroup;
+ int filetype = cft->private;
+ char *buffer, *b;
+ int retval = 0, count;
+ struct dev_whitelist_item wh;
+
+ if (!capable(CAP_SYS_ADMIN))
+  return -EPERM;
+
+ devcgroup = cgroup_to_devcgroup(cgroup);
+ cur_cgroup = task_cgroup(current, devices_subsys.subsys_id);
+ cur_devcgroup = cgroup_to_devcgroup(cur_cgroup);
+
+ buffer = kmalloc(nbytes+1, GFP_KERNEL);
+ if (!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));
+ b = buffer;
+
+ switch (*b) {
+ case 'a':
+  wh.type = DEV_ALL;
+  wh.access = ACC_MASK;
+  goto handle;
+ case 'b':
+  wh.type = DEV_BLOCK;
```

```
+  break;
+ case 'c':
+  wh.type = DEV_CHAR;
+  break;
+ default:
+  retval = -EINVAL;
+  goto out2;
+ }
+ b += 2;
+ if (*b == '*') {
+  wh.major = ~0;
+  b++;
+ } else if (isdigit(*b)) {
+  wh.major = 0;
+  while (isdigit(*b)) {
+   wh.major = wh.major*10+(*b-'0');
+   b++;
+  }
+ } else {
+  retval = -EINVAL;
+  goto out2;
+ }
+ if (*b != ':') {
+  retval = -EINVAL;
+  goto out2;
+ }
+ b++;
+
+ /* read minor */
+ if (*b == '*') {
+  wh.minor = ~0;
+  b++;
+ } else if (isdigit(*b)) {
+  wh.minor = 0;
+  while (isdigit(*b)) {
+   wh.minor = wh.minor*10+(*b-'0');
+   b++;
+  }
+ } else {
+  retval = -EINVAL;
+  goto out2;
+ }
+ if (!isspace(*b)) {
+  retval = -EINVAL;
+  goto out2;
+ }
+ for (b++, count = 0; count < 3; count++, b++) {
+  switch (*b) {
```

```
+  case 'r':
+   wh.access |= ACC_READ;
+   break;
+  case 'w':
+   wh.access |= ACC_WRITE;
+   break;
+  case 'm':
+   wh.access |= ACC_MKNOD;
+   break;
+  case '\n':
+  case '\0':
+   break;
+  default:
+   retval = -EINVAL;
+   goto out2;
+  }
+ }
+
+handle:
+ retval = 0;
+ switch (filetype) {
+ case DEVCG_ALLOW:
+  if (!parent_has_perm(cgroup, &wh))
+   retval = -EPERM;
+  else
+   retval = dev_whitelist_add(devcgroup, &wh);
+  break;
+ case DEVCG_DENY:
+  dev_whitelist_rm(devcgroup, &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 = devcgroup_access_read,
+  .write  = devcgroup_access_write,
+  .private = DEVCG_ALLOW,
+ },
+ {
+  .name = "deny",
+  .write = devcgroup_access_write,
+  .private = DEVCG_DENY,
+ },
+};
+
+static int devcgroup_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 devices_subsys = {
+ .name = "devices",
+ .can_attach = devcgroup_can_attach,
+ .create = devcgroup_create,
+ .destroy  = devcgroup_destroy,
+ .populate = devcgroup_populate,
+ .subsys_id = devices_subsys_id,
+};
+
+int devcgroup_inode_permission(struct inode *inode, int mask)
+{
+ 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, devices_subsys.subsys_id);
+ dev_cgroup = cgroup_to_devcgroup(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 acc_check;
+  if ((wh->type & DEV_BLOCK) && !S_ISBLK(inode->i_mode))
```

```
+   continue;
+  if ((wh->type & DEV_CHAR) && !S_ISCHR(inode->i_mode))
+   continue;
+  if (wh->major != ~0 && wh->major != imajor(inode))
+   continue;
+  if (wh->minor != ~0 && 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);
+
+ return -EPERM;
+}
+
+int devcgroup_inode_mknod(int mode, dev_t dev)
+{
+ struct cgroup *cgroup;
+ struct dev_cgroup *dev_cgroup;
+ struct dev_whitelist_item *wh;
+
+ cgroup = task_cgroup(current, devices_subsys.subsys_id);
+ dev_cgroup = cgroup_to_devcgroup(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 acc_check;
+  if ((wh->type & DEV_BLOCK) && !S_ISBLK(mode))
+   continue;
+  if ((wh->type & DEV_CHAR) && !S_ISCHR(mode))
+   continue;
+  if (wh->major != ~0 && wh->major != MAJOR(dev))
+   continue;
+  if (wh->minor != ~0 && wh->minor != MINOR(dev))
+   continue;
+acc_check:
+  if (!(wh->access & ACC_MKNOD))
+   continue;
+  spin_unlock(&dev_cgroup->lock);
+  return 0;
```

+ }
+ spin_unlock(&dev_cgroup->lock);
+ return -EPERM;
+}
--
1.5.1.1.GIT

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

---

Subject: Re: [PATCH] cgroups: implement device whitelist (v4)
Posted by James Morris on Tue, 18 Mar 2008 04:17:53 GMT
View Forum Message <> Reply to Message

On Mon, 17 Mar 2008, Serge E. Hallyn wrote:

> Implement a cgroup to track and enforce open and mknod restrictions on device
> files. A device cgroup associates a device access whitelist with each
> cgroup. A whitelist entry has 4 fields. 'type' is a (all), c (char), or
> b (block). 'all' means it applies to all types and all major and minor
> numbers. Major and minor are either an integer or * for all.
> Access is a composition of r (read), w (write), and m (mknod).

Acked-by: James Morris <jmorris@namei.org>

--
James Morris
<jmorris@namei.org>

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

---

Subject: Re: [PATCH] cgroups: implement device whitelist (v4)
Posted by Li Zefan on Tue, 18 Mar 2008 05:15:05 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Implement a cgroup to track and enforce open and mknod restrictions on device
> files. A device cgroup associates a device access whitelist with each
> cgroup. A whitelist entry has 4 fields. 'type' is a (all), c (char), or

> b (block). 'all' means it applies to all types and all major and minor
> numbers. Major and minor are either an integer or * for all.
> Access is a composition of r (read), w (write), and m (mknod).
>
> The root device cgroup starts with rwm to 'all'. A child devcg gets
> a copy of the parent. Admins can then remove devices from the
> whitelist or add new entries. A child cgroup can never receive a
> device access which is denied its parent. However when a device
> access is removed from a parent it will not also be removed from the
> child(ren).
>
> An entry is added using devices.allow, and removed using
> devices.deny. For instance
>
> echo 'c 1:3 mr' > /cgroups/1/devices.allow
>
> allows cgroup 1 to read and mknod the device usually known as
> /dev/null. Doing
>
> echo a > /cgroups/1/devices.deny
>
> will remove the default 'a *:* mrw' entry.
>
> CAP_SYS_ADMIN is needed to change permissions or move another task
> to a new cgroup. A cgroup may not be granted more permissions than
> the cgroup's parent has. Any task can move itself between cgroups.
> This won't be sufficient, but we can decide the best way to
> adequately restrict movement later.
>
> The parsing of devices.allow/deny needs to be cleaned up a bit and
> Documented. I'd like to get an idea whether this approach is otherwise
> acceptable.
>
> Changelog:
> Mar 17 2008: Place specific device cgroup hooks next to
>   security_inode_{mknod,permission} rather than using
>   the security hooks.
>   Also remove most of the controls over tasks moving
>   between cgroups and playing with the allow and deny
>   permissions.
>   Switch to major:minor format.
>   Rename devcg to 'devices' to conform to cgroup naming.
> Mar 13 2008: move the dev_cgroup support into
>   capability hooks instead of having it
>   as a separate security module.
>   Support root_plug with devcg.
>   Note that due to this change, devcg will
>   not be enforcing if the dummy module is

```
>   loaded, or if selinux is loaded without
>   capabilities.
>  Mar 12 2008: allow dev_cgroup lsm to be used when
>   SECURITY=n, and allow stacking with SELinux
>   and Smack.  Don't work too hard in Kconfig
>   to prevent a warning when smack+devcg are
>   both compiled in, worry about that later.
>
```

I would like to give some comments in the code. :)

```
> Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
> ---
>  fs/namei.c                    |    9 +
>  include/linux/cgroup_subsys.h |    6 +
>  include/linux/device_cgroup.h |   12 +
>  init/Kconfig                  |    7 +
>  security/Makefile             |    1 +
>  security/device_cgroup.c      |  597 ++++++++++++++++++++++++++++++++++++++++++
>  6 files changed, 632 insertions(+), 0 deletions(-)
>  create mode 100644 include/linux/device_cgroup.h
>  create mode 100644 security/device_cgroup.c
>
> diff --git a/fs/namei.c b/fs/namei.c
> index dfb3cb8..6caed32 100644
> --- a/fs/namei.c
> +++ b/fs/namei.c
> @@ -30,6 +30,7 @@
>  #include <linux/capability.h>
>  #include <linux/file.h>
>  #include <linux/fcntl.h>
> +#include <linux/device_cgroup.h>
>  #include <asm/namei.h>
>  #include <asm/uaccess.h>
>
> @@ -281,6 +282,10 @@ int permission(struct inode *inode, int mask, struct nameidata *nd)
>   if (retval)
>    return retval;
>
> + retval = devcgroup_inode_permission(inode, mask);
> + if (retval)
> +  return retval;
> +
>   return security_inode_permission(inode, mask, nd);
>  }
>
> @@ -2028,6 +2033,10 @@ int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode,
dev_t dev)
```

>   if (!dir->i_op || !dir->i_op->mknod)
>   return -EPERM;
>
> + error = devcgroup_inode_mknod(mode, dev);
> + if (error)
> + return error;
> +
>   error = security_inode_mknod(dir, dentry, mode, dev);
>   if (error)
>   return error;
> diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
> index 1ddebfc..e287745 100644
> --- a/include/linux/cgroup_subsys.h
> +++ b/include/linux/cgroup_subsys.h
> @@ -42,3 +42,9 @@ SUBSYS(mem_cgroup)
> #endif
>
> /* */
> +
> +#ifdef CONFIG_CGROUP_DEVICE
> +SUBSYS(devices)
> +#endif
> +
> +/* */
> diff --git a/include/linux/device_cgroup.h b/include/linux/device_cgroup.h
> new file mode 100644
> index 0000000..0b0d9c3
> --- /dev/null
> +++ b/include/linux/device_cgroup.h
> @@ -0,0 +1,12 @@
> +#include <linux/module.h>
> +#include <linux/fs.h>
> +
> +#ifdef CONFIG_CGROUP_DEVICE
> +extern int devcgroup_inode_permission(struct inode *inode, int mask);
> +extern int devcgroup_inode_mknod(int mode, dev_t dev);
> +#else
> +static inline int devcgroup_inode_permission(struct inode *inode, int mask)
> +{ return 0; }
> +static inline int devcgroup_inode_mknod(int mode, dev_t dev)
> +{ return 0; }
> +#endif
> diff --git a/init/Kconfig b/init/Kconfig
> index 009f2d8..30868cd 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -298,6 +298,13 @@ config CGROUP_NS
>   for instance virtual servers and checkpoint/restart

>   jobs.
>
> +config CGROUP_DEVICE
> + bool "Device controller for cgroups"
> + depends on CGROUPS && EXPERIMENTAL
> + help
> +   Provides a cgroup implementing whitelists for devices which
> +   a process in the cgroup can mknod or open.
> +
> config CPUSETS
>   bool "Cpuset support"
>   depends on SMP && CGROUPS
> diff --git a/security/Makefile b/security/Makefile
> index 9e8b025..7ef1107 100644
> --- a/security/Makefile
> +++ b/security/Makefile
> @@ -18,3 +18,4 @@ obj-$(CONFIG_SECURITY_SELINUX) += selinux/built-in.o
> obj-$(CONFIG_SECURITY_SMACK) += commoncap.o smack/built-in.o
> obj-$(CONFIG_SECURITY_CAPABILITIES) += commoncap.o capability.o
> obj-$(CONFIG_SECURITY_ROOTPLUG) += commoncap.o root_plug.o
> +obj-$(CONFIG_CGROUP_DEVICE) += device_cgroup.o
> diff --git a/security/device_cgroup.c b/security/device_cgroup.c
> new file mode 100644
> index 0000000..33d8fd8
> --- /dev/null
> +++ b/security/device_cgroup.c
> @@ -0,0 +1,597 @@
> +/*
> + * dev_cgroup.c - device cgroup subsystem
> + *
> + * Copyright 2007 IBM Corp
> + */
> +
> +#include <linux/device_cgroup.h>
> +#include <linux/cgroup.h>
> +#include <linux/ctype.h>
> +#include <linux/list.h>
> +#include <asm/uaccess.h>
> +
> +#define ACC_MKNOD 1
> +#define ACC_READ 2
> +#define ACC_WRITE 4
> +#define ACC_MASK (ACC_MKNOD | ACC_READ | ACC_WRITE)
> +
> +#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.

When you say cgroup->lock, you mean dev_cgroup->lock, right?
So would it be better to make it clear in the comment?

> + * 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 kmalloc, 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;
> +};
> +
> +static inline struct dev_cgroup *cgroup_to_devcgroup(
> + struct cgroup *cgroup)
> +{
> + return container_of(cgroup_subsys_state(cgroup, devices_subsys_id),
> +   struct dev_cgroup, css);
> +}
> +
> +
> +struct cgroup_subsys devices_subsys;
> +
> +static int devcgroup_can_attach(struct cgroup_subsys *ss,
> + struct cgroup *new_cgroup, struct task_struct *task)
> +{

> +

redundant empty line

> + if (current != task && !capable(CAP_SYS_ADMIN))
> +   return -EPERM;
> +
> + return 0;
> +}
> +
> +/*
> + * called under cgroup_lock()
> + */
> +int dev_whitelist_copy(struct list_head *dest, struct list_head *orig)

static int

> +{
> + 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 dev_whitelist_addother tasks, we need the spinlock also
> + */
> +int dev_whitelist_add(struct dev_cgroup *dev_cgroup,
> +   struct dev_whitelist_item *wh)

ditoo

```
> +{
> + struct dev_whitelist_item *whcopy;
> +
> + whcopy = kmalloc(sizeof(*whcopy), GFP_KERNEL);
> + if (!whcopy)
> +  return -ENOMEM;
> +
> + memcpy(whcopy, wh, sizeof(*whcopy));
> + spin_lock(&dev_cgroup->lock);
> + list_add_tail(&whcopy->list, &dev_cgroup->whitelist);
> + spin_unlock(&dev_cgroup->lock);
> + return 0;
> +}
> +
> +/*
> + * 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)
```

ditto

```
> +{
> + 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)
> +   goto remove;
> +  if (walk->type != wh->type)
> +   continue;
> +  if (walk->major != ~0 && walk->major != wh->major)
> +   continue;
> +  if (walk->minor != ~0 && walk->minor != wh->minor)
> +   continue;
> +
> +remove:
> +  walk->access &= ~wh->access;
> +  if (!walk->access) {
> +   list_del(&walk->list);
> +   kfree(walk);
> +  }
> + }
> + spin_unlock(&dev_cgroup->lock);
```

```
> +}
> +
> +/*
> + * called from kernel/cgroup.c with cgroup_lock() held.
> + */
> +static struct cgroup_subsys_state *devcgroup_create(struct cgroup_subsys *ss,
> +      struct cgroup *cgroup)
> +{
> + struct dev_cgroup *dev_cgroup, *parent_dev_cgroup;
> + struct cgroup *parent_cgroup;
> + int ret;
> +
> + 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);

if (wh == NULL) ..

> +  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_devcgroup(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 devcgroup_destroy(struct cgroup_subsys *ss,
> +   struct cgroup *cgroup)
> +{
> + struct dev_cgroup *dev_cgroup;
> + struct dev_whitelist_item *wh, *tmp;
> +
```

```
> + dev_cgroup = cgroup_to_devcgroup(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)

static

> +{
> + 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)

static

> +{
> + if (type == DEV_ALL)
> +  return 'a';
> + if (type == DEV_CHAR)
> +  return 'c';
> + if (type == DEV_BLOCK)
> +  return 'b';
> + return 'X';
> +}
> +
> +static void set_majmin(char *str, int len, unsigned m)
> +{
> + memset(str, 0, len);
> + if (m == ~0)
> +  sprintf(str, "*");
> + else
> +  snprintf(str, len, "%d", m);
> +}
```

```
> +
> +char *print_whitelist(struct dev_cgroup *devcgroup, int *len)

static

> +{
> + char *buf, *s, acc[4];
> + struct dev_whitelist_item *wh;
> + int ret;
> + int count = 0;
> + char maj[10], min[10];
> +
> + 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);
> +  set_majmin(maj, 10, wh->major);
> +  set_majmin(min, 10, wh->minor);
> +  ret = snprintf(s, 4095-(s-buf), "%c %s:%s %s\n",
> +   type_to_char(wh->type), maj, min, 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 devcgroup_access_read(struct cgroup *cgroup,
> +   struct cftype *cft, struct file *file,
> +   char __user *userbuf, size_t nbytes, loff_t *ppos)
> +{
> + struct dev_cgroup *devcgroup = cgroup_to_devcgroup(cgroup);
> + int filetype = cft->private;
> + char *buffer;
> + int len, retval;
```

```
> +
> + if (filetype != DEVCG_ALLOW)
> +  return -EINVAL;
> + buffer = print_whitelist(devcgroup, &len);
> + if (IS_ERR(buffer))
> +  return PTR_ERR(buffer);
> +
> + retval = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, len);
> + kfree(buffer);
> + return retval;
> +}
> +
> +/*
> + * may_access_whitelist:
> + * does the access granted to dev_cgroup c contain the access
> + * requested in whitelist item refwh.
> + * return 1 if yes, 0 if no.
> + * call with c->lock held
> + */
> +static int may_access_whitelist(struct dev_cgroup *c,
> +           struct dev_whitelist_item *refwh)
> +{
> + struct dev_whitelist_item *whitem;
> +
> + list_for_each_entry(whitem, &c->whitelist, list) {
> +  if (whitem->type & DEV_ALL)
> +   return 1;
> +  if ((refwh->type & DEV_BLOCK) && !(whitem->type & DEV_BLOCK))
> +   continue;
> +  if ((refwh->type & DEV_CHAR) && !(whitem->type & DEV_CHAR))
> +   continue;
> +  if (whitem->major != ~0 && whitem->major != refwh->major)
> +   continue;
> +  if (whitem->minor != ~0 && whitem->minor != refwh->minor)
> +   continue;
> +  if (refwh->access & (~(whitem->access | ACC_MASK)))
> +   continue;
> +  return 1;
> + }
> + return 0;
> +}
> +
> +/*
> + * parent_has_perm:
> + * when adding a new allow rule to a device whitelist, the rule
> + * must be allowed in the parent device
> + */
> +static int parent_has_perm(struct cgroup *childcg,
```

```
> +      struct dev_whitelist_item *wh)
> +{
> + struct cgroup *pcg = childcg->parent;
> + struct dev_cgroup *parent;
> + int ret;
> +
> + if (!pcg)
> +  return 1;
> + parent = cgroup_to_devcgroup(pcg);
> + spin_lock(&parent->lock);
> + ret = may_access_whitelist(parent, wh);
> + spin_unlock(&parent->lock);
> + return ret;
> +}
> +
> +/*
> + * Modify the whitelist using allow/deny rules.
> + * CAP_SYS_ADMIN is needed for this.  It's at least separate from CAP_MKNOD
> + * so we can give a container CAP_MKNOD to let it create devices but not
> + * modify the whitelist.
> + * It seems likely we'll want to add a CAP_CONTAINER capability to allow
> + * us to also grant CAP_SYS_ADMIN to containers without giving away the
> + * device whitelist controls, but for now we'll stick with CAP_SYS_ADMIN
> + *
> + * Taking rules away is always allowed (given CAP_SYS_ADMIN).  Granting
> + * new access is only allowed if you're in the top-level cgroup, or your
> + * parent cgroup has the access you're asking for.
> + */
> +static ssize_t devcgroup_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 *devcgroup, *cur_devcgroup;
> + int filetype = cft->private;
> + char *buffer, *b;
> + int retval = 0, count;
> + struct dev_whitelist_item wh;
> +
> + if (!capable(CAP_SYS_ADMIN))
> +  return -EPERM;
> +
> + devcgroup = cgroup_to_devcgroup(cgroup);
> + cur_cgroup = task_cgroup(current, devices_subsys.subsys_id);
> + cur_devcgroup = cgroup_to_devcgroup(cur_cgroup);
> +
> + buffer = kmalloc(nbytes+1, GFP_KERNEL);
> + if (!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));
> + b = buffer;
> +
> + switch (*b) {
> + case 'a':
> +  wh.type = DEV_ALL;
> +  wh.access = ACC_MASK;
> +  goto handle;
> + case 'b':
> +  wh.type = DEV_BLOCK;
> +  break;
> + case 'c':
> +  wh.type = DEV_CHAR;
> +  break;
> + default:
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + b += 2;
> + if (*b == '*') {
> +  wh.major = ~0;
> +  b++;
> + } else if (isdigit(*b)) {
> +  wh.major = 0;
> +  while (isdigit(*b)) {
> +   wh.major = wh.major*10+(*b-'0');
> +   b++;
> +  }
> + } else {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + if (*b != ':') {
> +  retval = -EINVAL;
```

```
> +  goto out2;
> + }
> + b++;
> +
> + /* read minor */
> + if (*b == '*') {
> +  wh.minor = ~0;
> +  b++;
> + } else if (isdigit(*b)) {
> +  wh.minor = 0;
> +  while (isdigit(*b)) {
> +   wh.minor = wh.minor*10+(*b-'0');
> +   b++;
> +  }
> + } else {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + if (!isspace(*b)) {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + for (b++, count = 0; count < 3; count++, b++) {
> +  switch (*b) {
> +  case 'r':
> +   wh.access |= ACC_READ;
> +   break;
> +  case 'w':
> +   wh.access |= ACC_WRITE;
> +   break;
> +  case 'm':
> +   wh.access |= ACC_MKNOD;
> +   break;
> +  case '\n':
> +  case '\0':
> +   break;
> +  default:
> +   retval = -EINVAL;
> +   goto out2;
> +  }
> + }
> +
> +handle:
> + retval = 0;
> + switch (filetype) {
> + case DEVCG_ALLOW:
> +  if (!parent_has_perm(cgroup, &wh))
> +   retval = -EPERM;
```

```
> +  else
> +   retval = dev_whitelist_add(devcgroup, &wh);
> +  break;
> + case DEVCG_DENY:
> +  dev_whitelist_rm(devcgroup, &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 = devcgroup_access_read,
> +  .write  = devcgroup_access_write,
> +  .private = DEVCG_ALLOW,
> + },
> + {
> +  .name = "deny",
> +  .write = devcgroup_access_write,
> +  .private = DEVCG_DENY,
> + },
> +};
> +
> +static int devcgroup_populate(struct cgroup_subsys *ss,
> +     struct cgroup *cont)
```

could you use the name 'cgroup' or 'cgrp' instead of old name 'cont'?

```
> + .name = "devices",
> + .can_attach = devcgroup_can_attach,
> + .create = devcgroup_create,
> + .destroy  = devcgroup_destroy,
> + .populate = devcgroup_populate,
> + .subsys_id = devices_subsys_id,
> +};
```

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

Subject: Re: [PATCH] cgroups: implement device whitelist (v4)
Posted by Pavel Emelianov on Tue, 18 Mar 2008 07:57:03 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Implement a cgroup to track and enforce open and mknod restrictions on device
> files. A device cgroup associates a device access whitelist with each
> cgroup. A whitelist entry has 4 fields. 'type' is a (all), c (char), or
> b (block). 'all' means it applies to all types and all major and minor
> numbers. Major and minor are either an integer or * for all.
> Access is a composition of r (read), w (write), and m (mknod).
>
> The root device cgroup starts with rwm to 'all'. A child devcg gets
> a copy of the parent. Admins can then remove devices from the
> whitelist or add new entries. A child cgroup can never receive a
> device access which is denied its parent. However when a device
> access is removed from a parent it will not also be removed from the
> child(ren).
>
> An entry is added using devices.allow, and removed using
> devices.deny. For instance
>
> echo 'c 1:3 mr' > /cgroups/1/devices.allow
>
> allows cgroup 1 to read and mknod the device usually known as
> /dev/null. Doing
>
> echo a > /cgroups/1/devices.deny
>
> will remove the default 'a *:* mrw' entry.
>
> CAP_SYS_ADMIN is needed to change permissions or move another task
> to a new cgroup. A cgroup may not be granted more permissions than
> the cgroup's parent has. Any task can move itself between cgroups.
> This won't be sufficient, but we can decide the best way to
> adequately restrict movement later.
>
> The parsing of devices.allow/deny needs to be cleaned up a bit and
> Documented. I'd like to get an idea whether this approach is otherwise
> acceptable.
>
> Changelog:

> Mar 17 2008: Place specific device cgroup hooks next to
> security_inode_{mknod,permission} rather than using
> the security hooks.
> Also remove most of the controls over tasks moving
> between cgroups and playing with the allow and deny
> permissions.
> Switch to major:minor format.
> Rename devcg to 'devices' to conform to cgroup naming.
> Mar 13 2008: move the dev_cgroup support into
> capability hooks instead of having it
> as a separate security module.
> Support root_plug with devcg.
> Note that due to this change, devcg will
> not be enforcing if the dummy module is
> loaded, or if selinux is loaded without
> capabilities.
> Mar 12 2008: allow dev_cgroup lsm to be used when
> SECURITY=n, and allow stacking with SELinux
> and Smack. Don't work too hard in Kconfig
> to prevent a warning when smack+devcg are
> both compiled in, worry about that later.
>
> Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>

Looks-good-to: Pavel Emelyanov <xemul@openvz.org>

> ---
> fs/namei.c | 9 +
> include/linux/cgroup_subsys.h | 6 +
> include/linux/device_cgroup.h | 12 +
> init/Kconfig | 7 +
> security/Makefile | 1 +
> security/device_cgroup.c | 597 ++
> 6 files changed, 632 insertions(+), 0 deletions(-)
> create mode 100644 include/linux/device_cgroup.h
> create mode 100644 security/device_cgroup.c
>
> diff --git a/fs/namei.c b/fs/namei.c
> index dfb3cb8..6caed32 100644
> --- a/fs/namei.c
> +++ b/fs/namei.c
> @@ -30,6 +30,7 @@
> #include <linux/capability.h>
> #include <linux/file.h>
> #include <linux/fcntl.h>
> +#include <linux/device_cgroup.h>
> #include <asm/namei.h>
> #include <asm/uaccess.h>

>
> @@ -281,6 +282,10 @@ int permission(struct inode *inode, int mask, struct nameidata *nd)
>   if (retval)
>   return retval;
>
> + retval = devcgroup_inode_permission(inode, mask);
> + if (retval)
> + return retval;
> +
>   return security_inode_permission(inode, mask, nd);
> }
>
> @@ -2028,6 +2033,10 @@ int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode,
dev_t dev)
>   if (!dir->i_op || !dir->i_op->mknod)
>   return -EPERM;
>
> + error = devcgroup_inode_mknod(mode, dev);
> + if (error)
> + return error;
> +
>   error = security_inode_mknod(dir, dentry, mode, dev);
>   if (error)
>   return error;
> diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
> index 1ddebfc..e287745 100644
> --- a/include/linux/cgroup_subsys.h
> +++ b/include/linux/cgroup_subsys.h
> @@ -42,3 +42,9 @@ SUBSYS(mem_cgroup)
> #endif
>
> /* */
> +
> +#ifdef CONFIG_CGROUP_DEVICE
> +SUBSYS(devices)
> +#endif
> +
> +/* */
> diff --git a/include/linux/device_cgroup.h b/include/linux/device_cgroup.h
> new file mode 100644
> index 0000000..0b0d9c3
> --- /dev/null
> +++ b/include/linux/device_cgroup.h
> @@ -0,0 +1,12 @@
> +#include <linux/module.h>
> +#include <linux/fs.h>
> +
> +#ifdef CONFIG_CGROUP_DEVICE

> +extern int devcgroup_inode_permission(struct inode *inode, int mask);
> +extern int devcgroup_inode_mknod(int mode, dev_t dev);
> +#else
> +static inline int devcgroup_inode_permission(struct inode *inode, int mask)
> +{ return 0; }
> +static inline int devcgroup_inode_mknod(int mode, dev_t dev)
> +{ return 0; }
> +#endif
> diff --git a/init/Kconfig b/init/Kconfig
> index 009f2d8..30868cd 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -298,6 +298,13 @@ config CGROUP_NS
> for instance virtual servers and checkpoint/restart
> jobs.
>
> +config CGROUP_DEVICE
> + bool "Device controller for cgroups"
> + depends on CGROUPS && EXPERIMENTAL
> + help
> + Provides a cgroup implementing whitelists for devices which
> + a process in the cgroup can mknod or open.
> +
> config CPUSETS
> bool "Cpuset support"
> depends on SMP && CGROUPS
> diff --git a/security/Makefile b/security/Makefile
> index 9e8b025..7ef1107 100644
> --- a/security/Makefile
> +++ b/security/Makefile
> @@ -18,3 +18,4 @@ obj-$(CONFIG_SECURITY_SELINUX) += selinux/built-in.o
> obj-$(CONFIG_SECURITY_SMACK) += commoncap.o smack/built-in.o
> obj-$(CONFIG_SECURITY_CAPABILITIES) += commoncap.o capability.o
> obj-$(CONFIG_SECURITY_ROOTPLUG) += commoncap.o root_plug.o
> +obj-$(CONFIG_CGROUP_DEVICE) += device_cgroup.o
> diff --git a/security/device_cgroup.c b/security/device_cgroup.c
> new file mode 100644
> index 0000000..33d8fd8
> --- /dev/null
> +++ b/security/device_cgroup.c
> @@ -0,0 +1,597 @@
> +/*
> + * dev_cgroup.c - device cgroup subsystem
> + *
> + * Copyright 2007 IBM Corp
> + */
> +
> +#include <linux/device_cgroup.h>

```
> +#include <linux/cgroup.h>
> +#include <linux/ctype.h>
> +#include <linux/list.h>
> +#include <asm/uaccess.h>
> +
> +#define ACC_MKNOD 1
> +#define ACC_READ  2
> +#define ACC_WRITE 4
> +#define ACC_MASK (ACC_MKNOD | ACC_READ | ACC_WRITE)
> +
> +#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 kmalloc, 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;
> +};
> +
> +static inline struct dev_cgroup *cgroup_to_devcgroup(
> +  struct cgroup *cgroup)
> +{
```

```
> + return container_of(cgroup_subsys_state(cgroup, devices_subsys_id),
> +       struct dev_cgroup, css);
> +}
> +
> +
> +struct cgroup_subsys devices_subsys;
> +
> +static int devcgroup_can_attach(struct cgroup_subsys *ss,
> +  struct cgroup *new_cgroup, struct task_struct *task)
> +{
> +
> + if (current != task && !capable(CAP_SYS_ADMIN))
> +   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
> + */
> +int dev_whitelist_add(struct dev_cgroup *dev_cgroup,
> +    struct dev_whitelist_item *wh)
> +{
> + struct dev_whitelist_item *whcopy;
> +
> + whcopy = kmalloc(sizeof(*whcopy), GFP_KERNEL);
> + if (!whcopy)
> +  return -ENOMEM;
> +
> + memcpy(whcopy, wh, sizeof(*whcopy));
> + spin_lock(&dev_cgroup->lock);
> + list_add_tail(&whcopy->list, &dev_cgroup->whitelist);
> + spin_unlock(&dev_cgroup->lock);
> + return 0;
> +}
> +
> +/*
> + * 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)
> +   goto remove;
> +  if (walk->type != wh->type)
> +   continue;
> +  if (walk->major != ~0 && walk->major != wh->major)
> +   continue;
> +  if (walk->minor != ~0 && walk->minor != wh->minor)
> +   continue;
> +
> +remove:
> +  walk->access &= ~wh->access;
> +  if (!walk->access) {
> +   list_del(&walk->list);
> +   kfree(walk);
> +  }
> + }
> + spin_unlock(&dev_cgroup->lock);
> +}
> +
```

```
> +/*
> + * called from kernel/cgroup.c with cgroup_lock() held.
> + */
> +static struct cgroup_subsys_state *devcgroup_create(struct cgroup_subsys *ss,
> +      struct cgroup *cgroup)
> +{
> + struct dev_cgroup *dev_cgroup, *parent_dev_cgroup;
> + struct cgroup *parent_cgroup;
> + int ret;
> +
> + 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_devcgroup(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 devcgroup_destroy(struct cgroup_subsys *ss,
> +   struct cgroup *cgroup)
> +{
> + struct dev_cgroup *dev_cgroup;
> + struct dev_whitelist_item *wh, *tmp;
> +
> + dev_cgroup = cgroup_to_devcgroup(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';
> +}
> +
> +static void set_majmin(char *str, int len, unsigned m)
> +{
> + memset(str, 0, len);
> + if (m == ~0)
> +  sprintf(str, "*");
> + else
> +  snprintf(str, len, "%d", m);
> +}
> +
> +char *print_whitelist(struct dev_cgroup *devcgroup, int *len)
> +{
> + char *buf, *s, acc[4];
> + struct dev_whitelist_item *wh;
> + int ret;
> + int count = 0;
> + char maj[10], min[10];
> +
> + 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);
> +  set_majmin(maj, 10, wh->major);
> +  set_majmin(min, 10, wh->minor);
> +  ret = snprintf(s, 4095-(s-buf), "%c %s:%s %s\n",
> +   type_to_char(wh->type), maj, min, 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 devcgroup_access_read(struct cgroup *cgroup,
> +   struct cftype *cft, struct file *file,
> +   char __user *userbuf, size_t nbytes, loff_t *ppos)
> +{
> + struct dev_cgroup *devcgroup = cgroup_to_devcgroup(cgroup);
> + int filetype = cft->private;
> + char *buffer;
> + int len, retval;
> +
> + if (filetype != DEVCG_ALLOW)
> +  return -EINVAL;
> + buffer = print_whitelist(devcgroup, &len);
> + if (IS_ERR(buffer))
> +  return PTR_ERR(buffer);
> +
> + retval = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, len);
> + kfree(buffer);
> + return retval;
> +}
> +
> +/*
> + * may_access_whitelist:
```

```
> + * does the access granted to dev_cgroup c contain the access
> + * requested in whitelist item refwh.
> + * return 1 if yes, 0 if no.
> + * call with c->lock held
> + */
> +static int may_access_whitelist(struct dev_cgroup *c,
> +           struct dev_whitelist_item *refwh)
> +{
> + struct dev_whitelist_item *whitem;
> +
> + list_for_each_entry(whitem, &c->whitelist, list) {
> +  if (whitem->type & DEV_ALL)
> +   return 1;
> +  if ((refwh->type & DEV_BLOCK) && !(whitem->type & DEV_BLOCK))
> +   continue;
> +  if ((refwh->type & DEV_CHAR) && !(whitem->type & DEV_CHAR))
> +   continue;
> +  if (whitem->major != ~0 && whitem->major != refwh->major)
> +   continue;
> +  if (whitem->minor != ~0 && whitem->minor != refwh->minor)
> +   continue;
> +  if (refwh->access & (~(whitem->access | ACC_MASK)))
> +   continue;
> +  return 1;
> + }
> + return 0;
> +}
> +
> +/*
> + * parent_has_perm:
> + * when adding a new allow rule to a device whitelist, the rule
> + * must be allowed in the parent device
> + */
> +static int parent_has_perm(struct cgroup *childcg,
> +      struct dev_whitelist_item *wh)
> +{
> + struct cgroup *pcg = childcg->parent;
> + struct dev_cgroup *parent;
> + int ret;
> +
> + if (!pcg)
> +  return 1;
> + parent = cgroup_to_devcgroup(pcg);
> + spin_lock(&parent->lock);
> + ret = may_access_whitelist(parent, wh);
> + spin_unlock(&parent->lock);
> + return ret;
> +}
```

```
> +
> +/*
> + * Modify the whitelist using allow/deny rules.
> + * CAP_SYS_ADMIN is needed for this.  It's at least separate from CAP_MKNOD
> + * so we can give a container CAP_MKNOD to let it create devices but not
> + * modify the whitelist.
> + * It seems likely we'll want to add a CAP_CONTAINER capability to allow
> + * us to also grant CAP_SYS_ADMIN to containers without giving away the
> + * device whitelist controls, but for now we'll stick with CAP_SYS_ADMIN
> + *
> + * Taking rules away is always allowed (given CAP_SYS_ADMIN).  Granting
> + * new access is only allowed if you're in the top-level cgroup, or your
> + * parent cgroup has the access you're asking for.
> + */
> +static ssize_t devcgroup_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 *devcgroup, *cur_devcgroup;
> + int filetype = cft->private;
> + char *buffer, *b;
> + int retval = 0, count;
> + struct dev_whitelist_item wh;
> +
> + if (!capable(CAP_SYS_ADMIN))
> +  return -EPERM;
> +
> + devcgroup = cgroup_to_devcgroup(cgroup);
> + cur_cgroup = task_cgroup(current, devices_subsys.subsys_id);
> + cur_devcgroup = cgroup_to_devcgroup(cur_cgroup);
> +
> + buffer = kmalloc(nbytes+1, GFP_KERNEL);
> + if (!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));
> + b = buffer;
> +
> + switch (*b) {
> + case 'a':
> +  wh.type = DEV_ALL;
> +  wh.access = ACC_MASK;
> +  goto handle;
> + case 'b':
> +  wh.type = DEV_BLOCK;
> +  break;
> + case 'c':
> +  wh.type = DEV_CHAR;
> +  break;
> + default:
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + b += 2;
> + if (*b == '*') {
> +  wh.major = ~0;
> +  b++;
> + } else if (isdigit(*b)) {
> +  wh.major = 0;
> +  while (isdigit(*b)) {
> +   wh.major = wh.major*10+(*b-'0');
> +   b++;
> +  }
> + } else {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + if (*b != ':') {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + b++;
> +
> + /* read minor */
> + if (*b == '*') {
> +  wh.minor = ~0;
> +  b++;
> + } else if (isdigit(*b)) {
> +  wh.minor = 0;
> +  while (isdigit(*b)) {
> +   wh.minor = wh.minor*10+(*b-'0');
> +   b++;
> +  }
```

```
> + } else {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + if (!isspace(*b)) {
> +  retval = -EINVAL;
> +  goto out2;
> + }
> + for (b++, count = 0; count < 3; count++, b++) {
> +  switch (*b) {
> +  case 'r':
> +   wh.access |= ACC_READ;
> +   break;
> +  case 'w':
> +   wh.access |= ACC_WRITE;
> +   break;
> +  case 'm':
> +   wh.access |= ACC_MKNOD;
> +   break;
> +  case '\n':
> +  case '\0':
> +   break;
> +  default:
> +   retval = -EINVAL;
> +   goto out2;
> +  }
> + }
> +
> +handle:
> + retval = 0;
> + switch (filetype) {
> + case DEVCG_ALLOW:
> +  if (!parent_has_perm(cgroup, &wh))
> +   retval = -EPERM;
> +  else
> +   retval = dev_whitelist_add(devcgroup, &wh);
> +  break;
> + case DEVCG_DENY:
> +  dev_whitelist_rm(devcgroup, &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 = devcgroup_access_read,
> +  .write  = devcgroup_access_write,
> +  .private = DEVCG_ALLOW,
> + },
> + {
> +  .name = "deny",
> +  .write = devcgroup_access_write,
> +  .private = DEVCG_DENY,
> + },
> +};
> +
> +static int devcgroup_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 devices_subsys = {
> + .name = "devices",
> + .can_attach = devcgroup_can_attach,
> + .create = devcgroup_create,
> + .destroy  = devcgroup_destroy,
> + .populate = devcgroup_populate,
> + .subsys_id = devices_subsys_id,
> +};
> +
> +int devcgroup_inode_permission(struct inode *inode, int mask)
> +{
> + 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, devices_subsys.subsys_id);
> + dev_cgroup = cgroup_to_devcgroup(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 acc_check;
> +  if ((wh->type & DEV_BLOCK) && !S_ISBLK(inode->i_mode))
> +   continue;
> +  if ((wh->type & DEV_CHAR) && !S_ISCHR(inode->i_mode))
> +   continue;
> +  if (wh->major != ~0 && wh->major != imajor(inode))
> +   continue;
> +  if (wh->minor != ~0 && 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);
> +
> + return -EPERM;
> +}
> +
> +int devcgroup_inode_mknod(int mode, dev_t dev)
> +{
> + struct cgroup *cgroup;
> + struct dev_cgroup *dev_cgroup;
> + struct dev_whitelist_item *wh;
> +
> + cgroup = task_cgroup(current, devices_subsys.subsys_id);
> + dev_cgroup = cgroup_to_devcgroup(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 acc_check;
> +  if ((wh->type & DEV_BLOCK) && !S_ISBLK(mode))
> +   continue;
> +  if ((wh->type & DEV_CHAR) && !S_ISCHR(mode))
```

> +   continue;
> + if (wh->major != ~0 && wh->major != MAJOR(dev))
> +   continue;
> + if (wh->minor != ~0 && wh->minor != MINOR(dev))
> +   continue;
> +acc_check:
> + if (!(wh->access & ACC_MKNOD))
> +   continue;
> + spin_unlock(&dev_cgroup->lock);
> + return 0;
> + }
> + spin_unlock(&dev_cgroup->lock);
> + return -EPERM;
> +}

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

---

Subject: Re: [PATCH] cgroups: implement device whitelist (v4)
Posted by serue on Tue, 18 Mar 2008 14:10:45 GMT
View Forum Message <> Reply to Message

Quoting Li Zefan (lizf@cn.fujitsu.com):
> Serge E. Hallyn wrote:
> > Implement a cgroup to track and enforce open and mknod restrictions on device
> > files. A device cgroup associates a device access whitelist with each
> > cgroup. A whitelist entry has 4 fields. 'type' is a (all), c (char), or
> > b (block). 'all' means it applies to all types and all major and minor
> > numbers. Major and minor are either an integer or * for all.
> > Access is a composition of r (read), w (write), and m (mknod).
> >
> > The root device cgroup starts with rwm to 'all'. A child devcg gets
> > a copy of the parent. Admins can then remove devices from the
> > whitelist or add new entries. A child cgroup can never receive a
> > device access which is denied its parent. However when a device
> > access is removed from a parent it will not also be removed from the
> > child(ren).
> >
> > An entry is added using devices.allow, and removed using
> > devices.deny. For instance
> >
> > echo 'c 1:3 mr' > /cgroups/1/devices.allow
> >
> > allows cgroup 1 to read and mknod the device usually known as
> > /dev/null. Doing

> >
> > echo a > /cgroups/1/devices.deny
> >
> > will remove the default 'a *:* mrw' entry.
> >
> > CAP_SYS_ADMIN is needed to change permissions or move another task
> > to a new cgroup. A cgroup may not be granted more permissions than
> > the cgroup's parent has. Any task can move itself between cgroups.
> > This won't be sufficient, but we can decide the best way to
> > adequately restrict movement later.
> >
> > The parsing of devices.allow/deny needs to be cleaned up a bit and
> > Documented. I'd like to get an idea whether this approach is otherwise
> > acceptable.
> >
> > Changelog:
> > Mar 17 2008: Place specific device cgroup hooks next to
> >   security_inode_{mknod,permission} rather than using
> >   the security hooks.
> >   Also remove most of the controls over tasks moving
> >   between cgroups and playing with the allow and deny
> >   permissions.
> >   Switch to major:minor format.
> >   Rename devcg to 'devices' to conform to cgroup naming.
> > Mar 13 2008: move the dev_cgroup support into
> >   capability hooks instead of having it
> >   as a separate security module.
> >   Support root_plug with devcg.
> >   Note that due to this change, devcg will
> >   not be enforcing if the dummy module is
> >   loaded, or if selinux is loaded without
> >   capabilities.
> > Mar 12 2008: allow dev_cgroup lsm to be used when
> >   SECURITY=n, and allow stacking with SELinux
> >   and Smack. Don't work too hard in Kconfig
> >   to prevent a warning when smack+devcg are
> >   both compiled in, worry about that later.
> >
>
> I would like to give some comments in the code. :)

Thanks for taking a look.

> > Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
> > ---
> > fs/namei.c | 9 +
> > include/linux/cgroup_subsys.h | 6 +
> > include/linux/device_cgroup.h | 12 +

```
> >  init/Kconfig                 |    7 +
> >  security/Makefile            |    1 +
> >  security/device_cgroup.c     |  597 ++++++++++++++++++++++++++++++++++++++++
> >  6 files changed, 632 insertions(+), 0 deletions(-)
> >  create mode 100644 include/linux/device_cgroup.h
> >  create mode 100644 security/device_cgroup.c
> >
> > diff --git a/fs/namei.c b/fs/namei.c
> > index dfb3cb8..6caed32 100644
> > --- a/fs/namei.c
> > +++ b/fs/namei.c
> > @@ -30,6 +30,7 @@
> >  #include <linux/capability.h>
> >  #include <linux/file.h>
> >  #include <linux/fcntl.h>
> > +#include <linux/device_cgroup.h>
> >  #include <asm/namei.h>
> >  #include <asm/uaccess.h>
> >
> > @@ -281,6 +282,10 @@ int permission(struct inode *inode, int mask, struct nameidata *nd)
> >   if (retval)
> >    return retval;
> >
> > + retval = devcgroup_inode_permission(inode, mask);
> > + if (retval)
> > +  return retval;
> > +
> >   return security_inode_permission(inode, mask, nd);
> >  }
> >
> > @@ -2028,6 +2033,10 @@ int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode,
dev_t dev)
> >   if (!dir->i_op || !dir->i_op->mknod)
> >    return -EPERM;
> >
> > + error = devcgroup_inode_mknod(mode, dev);
> > + if (error)
> > +  return error;
> > +
> >   error = security_inode_mknod(dir, dentry, mode, dev);
> >   if (error)
> >    return error;
> > diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
> > index 1ddebfc..e287745 100644
> > --- a/include/linux/cgroup_subsys.h
> > +++ b/include/linux/cgroup_subsys.h
> > @@ -42,3 +42,9 @@ SUBSYS(mem_cgroup)
> >  #endif
```

> >
> > /* */
> > +
> > +#ifdef CONFIG_CGROUP_DEVICE
> > +SUBSYS(devices)
> > +#endif
> > +
> > +/* */
> > diff --git a/include/linux/device_cgroup.h b/include/linux/device_cgroup.h
> > new file mode 100644
> > index 0000000..0b0d9c3
> > --- /dev/null
> > +++ b/include/linux/device_cgroup.h
> > @@ -0,0 +1,12 @@
> > +#include <linux/module.h>
> > +#include <linux/fs.h>
> > +
> > +#ifdef CONFIG_CGROUP_DEVICE
> > +extern int devcgroup_inode_permission(struct inode *inode, int mask);
> > +extern int devcgroup_inode_mknod(int mode, dev_t dev);
> > +#else
> > +static inline int devcgroup_inode_permission(struct inode *inode, int mask)
> > +{ return 0; }
> > +static inline int devcgroup_inode_mknod(int mode, dev_t dev)
> > +{ return 0; }
> > +#endif
> > diff --git a/init/Kconfig b/init/Kconfig
> > index 009f2d8..30868cd 100644
> > --- a/init/Kconfig
> > +++ b/init/Kconfig
> > @@ -298,6 +298,13 @@ config CGROUP_NS
> >   for instance virtual servers and checkpoint/restart
> >   jobs.
> >
> > +config CGROUP_DEVICE
> > + bool "Device controller for cgroups"
> > + depends on CGROUPS && EXPERIMENTAL
> > + help
> > +   Provides a cgroup implementing whitelists for devices which
> > +   a process in the cgroup can mknod or open.
> > +
> > config CPUSETS
> >   bool "Cpuset support"
> >   depends on SMP && CGROUPS
> > diff --git a/security/Makefile b/security/Makefile
> > index 9e8b025..7ef1107 100644
> > --- a/security/Makefile
> > +++ b/security/Makefile

> > @@ -18,3 +18,4 @@ obj-$(CONFIG_SECURITY_SELINUX) += selinux/built-in.o
> > obj-$(CONFIG_SECURITY_SMACK) += commoncap.o smack/built-in.o
> > obj-$(CONFIG_SECURITY_CAPABILITIES) += commoncap.o capability.o
> > obj-$(CONFIG_SECURITY_ROOTPLUG) += commoncap.o root_plug.o
> > +obj-$(CONFIG_CGROUP_DEVICE) += device_cgroup.o
> > diff --git a/security/device_cgroup.c b/security/device_cgroup.c
> > new file mode 100644
> > index 0000000..33d8fd8
> > --- /dev/null
> > +++ b/security/device_cgroup.c
> > @@ -0,0 +1,597 @@
> > +/*
> > + * dev_cgroup.c - device cgroup subsystem
> > + *
> > + * Copyright 2007 IBM Corp
> > + */
> > +
> > +#include <linux/device_cgroup.h>
> > +#include <linux/cgroup.h>
> > +#include <linux/ctype.h>
> > +#include <linux/list.h>
> > +#include <asm/uaccess.h>
> > +
> > +#define ACC_MKNOD 1
> > +#define ACC_READ 2
> > +#define ACC_WRITE 4
> > +#define ACC_MASK (ACC_MKNOD | ACC_READ | ACC_WRITE)
> > +
> > +#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.
>
> When you say cgroup->lock, you mean dev_cgroup->lock, right?
> So would it be better to make it clear in the comment?

Yes.

> > + * 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 kmalloc, 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;
> > +};
> > +
> > +static inline struct dev_cgroup *cgroup_to_devcgroup(
> > + struct cgroup *cgroup)
> > +{
> > + return container_of(cgroup_subsys_state(cgroup, devices_subsys_id),
> > +   struct dev_cgroup, css);
> > +}
> > +
> > +
> > +struct cgroup_subsys devices_subsys;
> > +
> > +static int devcgroup_can_attach(struct cgroup_subsys *ss,
> > + struct cgroup *new_cgroup, struct task_struct *task)
> > +{
> > +
>
> redundant empty line
>
> > + if (current != task && !capable(CAP_SYS_ADMIN))
> > +   return -EPERM;
> > +
> > + return 0;
> > +}
> > +
> > +/*
> > + * called under cgroup_lock()
> > + */
> > +int dev_whitelist_copy(struct list_head *dest, struct list_head *orig)

>
> static int
>
> > +{
> > + 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 dev_whitelist_addother tasks, we need the spinlock also
> > + */
> > +int dev_whitelist_add(struct dev_cgroup *dev_cgroup,
> > +   struct dev_whitelist_item *wh)
>
> ditoo
>
> > +{
> > + struct dev_whitelist_item *whcopy;
> > +
> > + whcopy = kmalloc(sizeof(*whcopy), GFP_KERNEL);
> > + if (!whcopy)
> > + return -ENOMEM;
> > +
> > + memcpy(whcopy, wh, sizeof(*whcopy));
> > + spin_lock(&dev_cgroup->lock);
> > + list_add_tail(&whcopy->list, &dev_cgroup->whitelist);
> > + spin_unlock(&dev_cgroup->lock);

```
> > + return 0;
> > +}
> > +
> > +/*
> > + * 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)
>
> ditto
>
> > +{
> > + 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)
> > +   goto remove;
> > +  if (walk->type != wh->type)
> > +   continue;
> > +  if (walk->major != ~0 && walk->major != wh->major)
> > +   continue;
> > +  if (walk->minor != ~0 && walk->minor != wh->minor)
> > +   continue;
> > +
> > +remove:
> > +  walk->access &= ~wh->access;
> > +  if (!walk->access) {
> > +   list_del(&walk->list);
> > +   kfree(walk);
> > +  }
> > + }
> > + spin_unlock(&dev_cgroup->lock);
> > +}
> > +
> > +/*
> > + * called from kernel/cgroup.c with cgroup_lock() held.
> > + */
> > +static struct cgroup_subsys_state *devcgroup_create(struct cgroup_subsys *ss,
> > +      struct cgroup *cgroup)
> > +{
> > + struct dev_cgroup *dev_cgroup, *parent_dev_cgroup;
> > + struct cgroup *parent_cgroup;
> > + int ret;
> > +
> > + 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);
>
> if (wh == NULL) ..

Egads. Over the weekend I had switched these to using a statically defined init_dev_cgroup and init_wh, but switched that back. Yes definately needs a check.

> > + 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_devcgroup(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 devcgroup_destroy(struct cgroup_subsys *ss,
> > +   struct cgroup *cgroup)
> > +{
> > + struct dev_cgroup *dev_cgroup;
> > + struct dev_whitelist_item *wh, *tmp;
> > +
> > + dev_cgroup = cgroup_to_devcgroup(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)
>
> static
>
> > +{
> > + 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)
>
> static
>
> > +{
> > + if (type == DEV_ALL)
> > +  return 'a';
> > + if (type == DEV_CHAR)
> > +  return 'c';
> > + if (type == DEV_BLOCK)
> > +  return 'b';
> > + return 'X';
> > +}
> > +
> > +static void set_majmin(char *str, int len, unsigned m)
> > +{
> > + memset(str, 0, len);
> > + if (m == ~0)
> > +  sprintf(str, "*");
> > + else
> > +  snprintf(str, len, "%d", m);
> > +}
> > +
> > +char *print_whitelist(struct dev_cgroup *devcgroup, int *len)
>
> static
>
> > +{
> > + char *buf, *s, acc[4];
> > + struct dev_whitelist_item *wh;
> > + int ret;
> > + int count = 0;
```

```
> > + char maj[10], min[10];
> > +
> > + 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);
> > +  set_majmin(maj, 10, wh->major);
> > +  set_majmin(min, 10, wh->minor);
> > +  ret = snprintf(s, 4095-(s-buf), "%c %s:%s %s\n",
> > +   type_to_char(wh->type), maj, min, 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 devcgroup_access_read(struct cgroup *cgroup,
> > +   struct cftype *cft, struct file *file,
> > +   char __user *userbuf, size_t nbytes, loff_t *ppos)
> > +{
> > + struct dev_cgroup *devcgroup = cgroup_to_devcgroup(cgroup);
> > + int filetype = cft->private;
> > + char *buffer;
> > + int len, retval;
> > +
> > + if (filetype != DEVCG_ALLOW)
> > +  return -EINVAL;
> > + buffer = print_whitelist(devcgroup, &len);
> > + if (IS_ERR(buffer))
> > +  return PTR_ERR(buffer);
> > +
> > + retval = simple_read_from_buffer(userbuf, nbytes, ppos, buffer, len);
> > + kfree(buffer);
> > + return retval;
```

```
> > +}
> > +
> > +/*
> > + * may_access_whitelist:
> > + * does the access granted to dev_cgroup c contain the access
> > + * requested in whitelist item refwh.
> > + * return 1 if yes, 0 if no.
> > + * call with c->lock held
> > + */
> > +static int may_access_whitelist(struct dev_cgroup *c,
> > +           struct dev_whitelist_item *refwh)
> > +{
> > + struct dev_whitelist_item *whitem;
> > +
> > + list_for_each_entry(whitem, &c->whitelist, list) {
> > +  if (whitem->type & DEV_ALL)
> > +   return 1;
> > +  if ((refwh->type & DEV_BLOCK) && !(whitem->type & DEV_BLOCK))
> > +   continue;
> > +  if ((refwh->type & DEV_CHAR) && !(whitem->type & DEV_CHAR))
> > +   continue;
> > +  if (whitem->major != ~0 && whitem->major != refwh->major)
> > +   continue;
> > +  if (whitem->minor != ~0 && whitem->minor != refwh->minor)
> > +   continue;
> > +  if (refwh->access & (~(whitem->access | ACC_MASK)))
> > +   continue;
> > +  return 1;
> > + }
> > + return 0;
> > +}
> > +
> > +/*
> > + * parent_has_perm:
> > + * when adding a new allow rule to a device whitelist, the rule
> > + * must be allowed in the parent device
> > + */
> > +static int parent_has_perm(struct cgroup *childcg,
> > +      struct dev_whitelist_item *wh)
> > +{
> > + struct cgroup *pcg = childcg->parent;
> > + struct dev_cgroup *parent;
> > + int ret;
> > +
> > + if (!pcg)
> > +  return 1;
> > + parent = cgroup_to_devcgroup(pcg);
> > + spin_lock(&parent->lock);
```

```
> > + ret = may_access_whitelist(parent, wh);
> > + spin_unlock(&parent->lock);
> > + return ret;
> > +}
> > +
> > +/*
> > + * Modify the whitelist using allow/deny rules.
> > + * CAP_SYS_ADMIN is needed for this.  It's at least separate from CAP_MKNOD
> > + * so we can give a container CAP_MKNOD to let it create devices but not
> > + * modify the whitelist.
> > + * It seems likely we'll want to add a CAP_CONTAINER capability to allow
> > + * us to also grant CAP_SYS_ADMIN to containers without giving away the
> > + * device whitelist controls, but for now we'll stick with CAP_SYS_ADMIN
> > + *
> > + * Taking rules away is always allowed (given CAP_SYS_ADMIN).  Granting
> > + * new access is only allowed if you're in the top-level cgroup, or your
> > + * parent cgroup has the access you're asking for.
> > + */
> > +static ssize_t devcgroup_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 *devcgroup, *cur_devcgroup;
> > + int filetype = cft->private;
> > + char *buffer, *b;
> > + int retval = 0, count;
> > + struct dev_whitelist_item wh;
> > +
> > + if (!capable(CAP_SYS_ADMIN))
> > +  return -EPERM;
> > +
> > + devcgroup = cgroup_to_devcgroup(cgroup);
> > + cur_cgroup = task_cgroup(current, devices_subsys.subsys_id);
> > + cur_devcgroup = cgroup_to_devcgroup(cur_cgroup);
> > +
> > + buffer = kmalloc(nbytes+1, GFP_KERNEL);
> > + if (!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));
> > + b = buffer;
> > +
> > + switch (*b) {
> > + case 'a':
> > +  wh.type = DEV_ALL;
> > +  wh.access = ACC_MASK;
> > +  goto handle;
> > + case 'b':
> > +  wh.type = DEV_BLOCK;
> > +  break;
> > + case 'c':
> > +  wh.type = DEV_CHAR;
> > +  break;
> > + default:
> > +  retval = -EINVAL;
> > +  goto out2;
> > + }
> > + b += 2;
> > + if (*b == '*') {
> > +  wh.major = ~0;
> > +  b++;
> > + } else if (isdigit(*b)) {
> > +  wh.major = 0;
> > +  while (isdigit(*b)) {
> > +   wh.major = wh.major*10+(*b-'0');
> > +   b++;
> > +  }
> > + } else {
> > +  retval = -EINVAL;
> > +  goto out2;
> > + }
> > + if (*b != ':') {
> > +  retval = -EINVAL;
> > +  goto out2;
> > + }
> > + b++;
> > +
> > + /* read minor */
> > + if (*b == '*') {
> > +  wh.minor = ~0;
> > +  b++;
> > + } else if (isdigit(*b)) {
> > +  wh.minor = 0;
```

```
> > +  while (isdigit(*b)) {
> > +   wh.minor = wh.minor*10+(*b-'0');
> > +   b++;
> > +  }
> > + } else {
> > +  retval = -EINVAL;
> > +  goto out2;
> > + }
> > + if (!isspace(*b)) {
> > +  retval = -EINVAL;
> > +  goto out2;
> > + }
> > + for (b++, count = 0; count < 3; count++, b++) {
> > +  switch (*b) {
> > +  case 'r':
> > +   wh.access |= ACC_READ;
> > +   break;
> > +  case 'w':
> > +   wh.access |= ACC_WRITE;
> > +   break;
> > +  case 'm':
> > +   wh.access |= ACC_MKNOD;
> > +   break;
> > +  case '\n':
> > +  case '\0':
> > +   break;
> > +  default:
> > +   retval = -EINVAL;
> > +   goto out2;
> > +  }
> > + }
> > +
> > +handle:
> > + retval = 0;
> > + switch (filetype) {
> > + case DEVCG_ALLOW:
> > +  if (!parent_has_perm(cgroup, &wh))
> > +   retval = -EPERM;
> > +  else
> > +   retval = dev_whitelist_add(devcgroup, &wh);
> > +  break;
> > + case DEVCG_DENY:
> > +  dev_whitelist_rm(devcgroup, &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 = devcgroup_access_read,
> > + .write = devcgroup_access_write,
> > + .private = DEVCG_ALLOW,
> > + },
> > + {
> > + .name = "deny",
> > + .write = devcgroup_access_write,
> > + .private = DEVCG_DENY,
> > + },
> > +};
> > +
> > +static int devcgroup_populate(struct cgroup_subsys *ss,
> > +   struct cgroup *cont)
>
> could you use the name 'cgroup' or 'cgrp' instead of old name 'cont'?

Sure. Certainly not cgrp, will use cgroup.

> > + .name = "devices",
> > + .can_attach = devcgroup_can_attach,
> > + .create = devcgroup_create,
> > + .destroy = devcgroup_destroy,
> > + .populate = devcgroup_populate,
> > + .subsys_id = devices_subsys_id,
> > +};

thanks,
-serge

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