Subject: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Mark Nelson on Tue, 27 Nov 2007 01:54:19 GMT
View Forum Message <> Reply to Message

Here's the latest version of sys_hijack.
Apologies for its lateness.

Thanks!

Mark.

Subject: [PATCH 1/2] namespaces: introduce sys_hijack (v10)

Move most of do_fork() into a new do_fork_task() which acts on
a new argument, task, rather than on current. do_fork() becomes
a call to do_fork_task(current, ...).

Introduce sys_hijack (for i386 and s390 only so far). It is like
clone, but in place of a stack pointer (which is assumed null) it
accepts a pid. The process identified by that pid is the one
which is actually cloned. Some state - including the file
table, the signals and sighand (and hence tty), and the ->parent
are taken from the calling process.

A process to be hijacked may be identified by process id, in the
case of HIJACK_PID. Alternatively, in the case of HIJACK_CG an
open fd for a cgroup 'tasks' file may be specified. The first
available task in that cgroup will then be hijacked.

HIJACK_NS is implemented as a third hijack method. The main
purpose is to allow entering an empty cgroup without having
to keep a task alive in the target cgroup. When HIJACK_NS
is called, only the cgroup and nsproxy are copied from the
cgroup. Security, user, and rootfs info is not retained
in the cgroups and so cannot be copied to the child task.

In order to hijack a process, the calling process must be
allowed to ptrace the target.

Sending sigstop to the hijacked task can trick its parent shell
(if it is a shell foreground task) into thinking it should retake
its tty.

So try not sending SIGSTOP, and instead hold the task_lock over
the hijacked task throughout the do_fork_task() operation.
This is really dangerous. I've fixed cgroup_fork() to not
task_lock(task) in the hijack case, but there may well be other
code called during fork which can under "some circumstances"

task_lock(task).

Still, this is working for me.

The effect is a sort of namespace enter. The following program
uses sys_hijack to 'enter' all namespaces of the specified task.
For instance in one terminal, do

```
 mount -t cgroup -ons cgroup /cgroup
 hostname
   qemu
 ns_exec -u /bin/sh
   hostname serge
         echo $$
           1073
   cat /proc/$$/cgroup
     ns:/node_1073
```

In another terminal then do

```
 hostname
   qemu
 cat /proc/$$/cgroup
   ns:/
 hijack pid 1073
   hostname
     serge
   cat /proc/$$/cgroup
     ns:/node_1073
 hijack cgroup /cgroup/node_1073/tasks
```

Changelog:
```
 Aug 23: send a stop signal to the hijacked process
  (like ptrace does).
 Oct 09: Update for 2.6.23-rc8-mm2 (mainly pidns)
  Don't take task_lock under rcu_read_lock
  Send hijacked process to cgroup_fork() as
  the first argument.
  Removed some unneeded task_locks.
 Oct 16: Fix bug introduced into alloc_pid.
 Oct 16: Add 'int which' argument to sys_hijack to
  allow later expansion to use cgroup in place
  of pid to specify what to hijack.
 Oct 24: Implement hijack by open cgroup file.
 Nov 02: Switch copying of task info: do full copy
  from current, then copy relevant pieces from
  hijacked task.
 Nov 06: Verbatim task_struct copy now comes from current,
```

after which copy_hijackable_taskinfo() copies
relevant context pieces from the hijack source.
Nov 07: Move arch-independent hijack code to kernel/fork.c
Nov 07: powerpc and x86_64 support (Mark Nelson)
Nov 07: Don't allow hijacking members of same session.
Nov 07: introduce cgroup_may_hijack, and may_hijack hook to
cgroup subsystems. The ns subsystem uses this to
enforce the rule that one may only hijack descendent
namespaces.
Nov 07: s390 support
Nov 08: don't send SIGSTOP to hijack source task
Nov 10: cache reference to nsproxy in ns cgroup for use in
hijacking an empty cgroup.
Nov 10: allow partial hijack of empty cgroup
Nov 13: don't double-get cgroup for hijack_ns
find_css_set() actually returns the set with a
reference already held, so cgroup_fork_fromcgroup()
by doing a get_css_set() was getting a second
reference. Therefore after exiting the hijack
task we could not rmdir the csgroup.
Nov 22: temporarily remove x86_64 and powerpc support
Nov 27: rebased on 2.6.24-rc3

```
============================================================
hijack.c
============================================================
/*
 * Your options are:
 * hijack pid 1078
 * hijack cgroup /cgroup/node_1078/tasks
 * hijack ns /cgroup/node_1078/tasks
 */

#define _BSD_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#if __i386__
#    define __NR_hijack  325
#elif __s390x__
```

```
#    define __NR_hijack  319
#else
#    error "Architecture not supported"
#endif

#ifndef CLONE_NEWUTS
#define CLONE_NEWUTS 0x04000000
#endif

void usage(char *me)
{
 printf("Usage: %s pid <pid>\n", me);
 printf("     | %s cgroup <cgroup_tasks_file>\n", me);
 printf("     | %s ns <cgroup_tasks_file>\n", me);
 exit(1);
}

int exec_shell(void)
{
 execl("/bin/sh", "/bin/sh", NULL);
}

#define HIJACK_PID 1
#define HIJACK_CG 2
#define HIJACK_NS 3

int main(int argc, char *argv[])
{
 int id;
 int ret;
 int status;
 int which_hijack;

 if (argc < 3 || !strcmp(argv[1], "-h"))
  usage(argv[0]);
 if (strcmp(argv[1], "cgroup") == 0)
  which_hijack = HIJACK_CG;
 else if (strcmp(argv[1], "ns") == 0)
  which_hijack = HIJACK_NS;
 else
  which_hijack = HIJACK_PID;

 switch(which_hijack) {
  case HIJACK_PID:
   id = atoi(argv[2]);
   printf("hijacking pid %d\n", id);
   break;
  case HIJACK_CG:
```

```
 case HIJACK_NS:
  id = open(argv[2], O_RDONLY);
  if (id == -1) {
   perror("cgroup open");
   return 1;
  }
  break;
 }

 ret = syscall(__NR_hijack, SIGCHLD, which_hijack, (unsigned long)id);

 if (which_hijack != HIJACK_PID)
  close(id);
 if  (ret == 0) {
  return exec_shell();
 } else if (ret < 0) {
  perror("sys_hijack");
 } else {
  printf("waiting on cloned process %d\n", ret);
  while(waitpid(-1, &status, __WALL) != -1)
    ;
  printf("cloned process exited with %d (waitpid ret %d)\n",
    status, ret);
 }

 return ret;
}
============================================================

Signed-off-by: Serge Hallyn <serue@us.ibm.com>
Signed-off-by: Mark Nelson <markn@au1.ibm.com>
---
 Documentation/cgroups.txt          |    9 +
 arch/s390/kernel/process.c         |   21 +++
 arch/x86/kernel/process_32.c       |   24 ++++
 arch/x86/kernel/syscall_table_32.S |    1
 include/asm-x86/unistd_32.h        |    3
 include/linux/cgroup.h             |   28 ++++-
 include/linux/nsproxy.h            |   12 +-
 include/linux/ptrace.h             |    1
 include/linux/sched.h              |   19 +++
 include/linux/syscalls.h           |    2
 kernel/cgroup.c                    |  133 ++++++++++++++++++++++-
 kernel/fork.c                      |  201 ++++++++++++++++++++++++++++++++++---
 kernel/ns_cgroup.c                 |   88 +++++++++++++++-
 kernel/nsproxy.c                   |    4
 kernel/ptrace.c                    |    7 +
 15 files changed, 523 insertions(+), 30 deletions(-)
```

```
Index: upstream/arch/s390/kernel/process.c
===================================================================
--- upstream.orig/arch/s390/kernel/process.c
+++ upstream/arch/s390/kernel/process.c
@@ -321,6 +321,27 @@ asmlinkage long sys_clone(void)
         parent_tidptr, child_tidptr);
 }

+asmlinkage long sys_hijack(void)
+{
+ struct pt_regs *regs = task_pt_regs(current);
+ unsigned long sp = regs->orig_gpr2;
+ unsigned long clone_flags = regs->gprs[3];
+ int which = regs->gprs[4];
+ unsigned int fd;
+ pid_t pid;
+
+ switch (which) {
+ case HIJACK_PID:
+  pid = regs->gprs[5];
+  return hijack_pid(pid, clone_flags, *regs, sp);
+ case HIJACK_CGROUP:
+  fd = (unsigned int) regs->gprs[5];
+  return hijack_cgroup(fd, clone_flags, *regs, sp);
+ default:
+  return -EINVAL;
+ }
+}
+
 /*
  * This is trivial, and on the face of it looks like it
  * could equally well be done in user mode.
Index: upstream/arch/x86/kernel/process_32.c
===================================================================
--- upstream.orig/arch/x86/kernel/process_32.c
+++ upstream/arch/x86/kernel/process_32.c
@@ -37,6 +37,7 @@
 #include <linux/personality.h>
 #include <linux/tick.h>
 #include <linux/percpu.h>
+#include <linux/cgroup.h>

 #include <asm/uaccess.h>
 #include <asm/pgtable.h>
@@ -781,6 +782,29 @@ asmlinkage int sys_clone(struct pt_regs
  return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
 }
```

```
+asmlinkage int sys_hijack(struct pt_regs regs)
+{
+ unsigned long sp = regs.esp;
+ unsigned long clone_flags = regs.ebx;
+ int which = regs.ecx;
+ unsigned int fd;
+ pid_t pid;
+
+ switch (which) {
+ case HIJACK_PID:
+  pid = regs.edx;
+  return hijack_pid(pid, clone_flags, regs, sp);
+ case HIJACK_CGROUP:
+  fd = (unsigned int) regs.edx;
+  return hijack_cgroup(fd, clone_flags, regs, sp);
+ case HIJACK_NS:
+  fd = (unsigned int) regs.edx;
+  return hijack_ns(fd, clone_flags, regs, sp);
+ default:
+  return -EINVAL;
+ }
+}
+
 /*
  * This is trivial, and on the face of it looks like it
  * could equally well be done in user mode.
Index: upstream/arch/x86/kernel/syscall_table_32.S
===================================================================
--- upstream.orig/arch/x86/kernel/syscall_table_32.S
+++ upstream/arch/x86/kernel/syscall_table_32.S
@@ -324,3 +324,4 @@ ENTRY(sys_call_table)
  .long sys_timerfd
  .long sys_eventfd
  .long sys_fallocate
+ .long sys_hijack  /* 325 */
Index: upstream/Documentation/cgroups.txt
===================================================================
--- upstream.orig/Documentation/cgroups.txt
+++ upstream/Documentation/cgroups.txt
@@ -495,6 +495,15 @@ LL=cgroup_mutex
 Called after the task has been attached to the cgroup, to allow any
 post-attachment activity that requires memory allocations or blocking.

+int may_hijack(struct cgroup_subsys *ss, struct cgroup *cont,
+       struct task_struct *task)
+LL=cgroup_mutex
+
```

```
+Called prior to hijacking a task.  Current is cloning a new child
+which is hijacking cgroup, namespace, and security context from
+the target task.  Called with the hijacked task locked.  Return
+0 to allow.
+
 void fork(struct cgroup_subsy *ss, struct task_struct *task)
 LL=callback_mutex, maybe read_lock(tasklist_lock)

Index: upstream/include/asm-x86/unistd_32.h
===================================================================
--- upstream.orig/include/asm-x86/unistd_32.h
+++ upstream/include/asm-x86/unistd_32.h
@@ -330,10 +330,11 @@
 #define __NR_timerfd  322
 #define __NR_eventfd  323
 #define __NR_fallocate  324
+#define __NR_hijack  325

 #ifdef __KERNEL__

-#define NR_syscalls 325
+#define NR_syscalls 326

 #define __ARCH_WANT_IPC_PARSE_VERSION
 #define __ARCH_WANT_OLD_READDIR
Index: upstream/include/linux/cgroup.h
===================================================================
--- upstream.orig/include/linux/cgroup.h
+++ upstream/include/linux/cgroup.h
@@ -14,19 +14,23 @@
 #include <linux/nodemask.h>
 #include <linux/rcupdate.h>
 #include <linux/cgroupstats.h>
+#include <linux/err.h>

 #ifdef CONFIG_CGROUPS

 struct cgroupfs_root;
 struct cgroup_subsys;
 struct inode;
+struct cgroup;

 extern int cgroup_init_early(void);
 extern int cgroup_init(void);
 extern void cgroup_init_smp(void);
 extern void cgroup_lock(void);
 extern void cgroup_unlock(void);
-extern void cgroup_fork(struct task_struct *p);
```

```
+extern void cgroup_fork(struct task_struct *parent, struct task_struct *p);
+extern void cgroup_fork_fromcgroup(struct cgroup *new_cg,
+      struct task_struct *child);
 extern void cgroup_fork_callbacks(struct task_struct *p);
 extern void cgroup_post_fork(struct task_struct *p);
 extern void cgroup_exit(struct task_struct *p, int run_callbacks);
@@ -236,6 +240,8 @@ struct cgroup_subsys {
  void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cont);
  int (*can_attach)(struct cgroup_subsys *ss,
      struct cgroup *cont, struct task_struct *tsk);
+ int (*may_hijack)(struct cgroup_subsys *ss,
+      struct cgroup *cont, struct task_struct *tsk);
  void (*attach)(struct cgroup_subsys *ss, struct cgroup *cont,
    struct cgroup *old_cont, struct task_struct *tsk);
  void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
@@ -304,12 +310,21 @@ struct task_struct *cgroup_iter_next(str
      struct cgroup_iter *it);
 void cgroup_iter_end(struct cgroup *cont, struct cgroup_iter *it);

+struct cgroup *cgroup_from_fd(unsigned int fd);
+struct task_struct *task_from_cgroup_fd(unsigned int fd);
+int cgroup_may_hijack(struct task_struct *tsk);
 #else /* !CONFIG_CGROUPS */
+struct cgroup {
+};

 static inline int cgroup_init_early(void) { return 0; }
 static inline int cgroup_init(void) { return 0; }
 static inline void cgroup_init_smp(void) {}
-static inline void cgroup_fork(struct task_struct *p) {}
+static inline void cgroup_fork(struct task_struct *parent,
+      struct task_struct *p) {}
+static inline void cgroup_fork_fromcgroup(struct cgroup *new_cg,
+      struct task_struct *child) {}
+
 static inline void cgroup_fork_callbacks(struct task_struct *p) {}
 static inline void cgroup_post_fork(struct task_struct *p) {}
 static inline void cgroup_exit(struct task_struct *p, int callbacks) {}
@@ -322,6 +337,15 @@ static inline int cgroupstats_build(stru
  return -EINVAL;
 }

+static inline struct cgroup *cgroup_from_fd(unsigned int fd) { return NULL; }
+static inline struct task_struct *task_from_cgroup_fd(unsigned int fd)
+{
+ return ERR_PTR(-EINVAL);
+}
+static inline int cgroup_may_hijack(struct task_struct *tsk)
```

```
+{
+ return 0;
+}
 #endif /* !CONFIG_CGROUPS */

 #endif /* _LINUX_CGROUP_H */
Index: upstream/include/linux/nsproxy.h
===================================================================
--- upstream.orig/include/linux/nsproxy.h
+++ upstream/include/linux/nsproxy.h
@@ -3,6 +3,7 @@

 #include <linux/spinlock.h>
 #include <linux/sched.h>
+#include <linux/err.h>

 struct mnt_namespace;
 struct uts_namespace;
@@ -81,10 +82,17 @@ static inline void get_nsproxy(struct ns
  atomic_inc(&ns->count);
 }

+struct cgroup;
 #ifdef CONFIG_CGROUP_NS
-int ns_cgroup_clone(struct task_struct *tsk);
+int ns_cgroup_clone(struct task_struct *tsk, struct nsproxy *nsproxy);
+int ns_cgroup_verify(struct cgroup *cgroup);
+void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup);
 #else
-static inline int ns_cgroup_clone(struct task_struct *tsk) { return 0; }
+static inline int ns_cgroup_clone(struct task_struct *tsk,
+ struct nsproxy *nsproxy) { return 0; }
+static inline int ns_cgroup_verify(struct cgroup *cgroup) { return 0; }
+static inline void copy_hijack_nsproxy(struct task_struct *tsk,
+          struct cgroup *cgroup) {}
 #endif

 #endif
Index: upstream/include/linux/ptrace.h
===================================================================
--- upstream.orig/include/linux/ptrace.h
+++ upstream/include/linux/ptrace.h
@@ -97,6 +97,7 @@ extern void __ptrace_link(struct task_st
 extern void __ptrace_unlink(struct task_struct *child);
 extern void ptrace_untrace(struct task_struct *child);
 extern int ptrace_may_attach(struct task_struct *task);
+extern int ptrace_may_attach_locked(struct task_struct *task);
```

```
 static inline void ptrace_link(struct task_struct *child,
           struct task_struct *new_parent)
Index: upstream/include/linux/sched.h
===================================================================
--- upstream.orig/include/linux/sched.h
+++ upstream/include/linux/sched.h
@@ -29,6 +29,13 @@
 #define CLONE_NEWNET  0x40000000 /* New network namespace */

 /*
+ * Hijack flags
+ */
+#define HIJACK_PID 1 /* 'id' is a pid */
+#define HIJACK_CGROUP 2 /* 'id' is an open fd for a cgroup dir */
+#define HIJACK_NS 3 /* 'id' is an open fd for a cgroup dir */
+
+/*
  * Scheduling policies
  */
 #define SCHED_NORMAL  0
@@ -1693,9 +1700,19 @@ extern int allow_signal(int);
 extern int disallow_signal(int);

 extern int do_execve(char *, char __user * __user *, char __user * __user *, struct pt_regs *);
-extern long do_fork(unsigned long, unsigned long, struct pt_regs *, unsigned long, int __user *,
int __user *);
+extern long do_fork(unsigned long, unsigned long, struct pt_regs *,
+  unsigned long, int __user *, int __user *);
 struct task_struct *fork_idle(int);

+extern int hijack_task(struct task_struct *task, unsigned long clone_flags,
+  struct pt_regs regs, unsigned long sp);
+extern int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
+  unsigned long sp);
+extern int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
+  struct pt_regs regs, unsigned long sp);
+extern int hijack_ns(unsigned int fd, unsigned long clone_flags,
+  struct pt_regs regs, unsigned long sp);
+
 extern void set_task_comm(struct task_struct *tsk, char *from);
 extern void get_task_comm(char *to, struct task_struct *tsk);

Index: upstream/include/linux/syscalls.h
===================================================================
--- upstream.orig/include/linux/syscalls.h
+++ upstream/include/linux/syscalls.h
@@ -614,4 +614,6 @@ asmlinkage long sys_fallocate(int fd, in
```

```
 int kernel_execve(const char *filename, char *const argv[], char *const envp[]);

+asmlinkage long sys_hijack(unsigned long flags, int which, unsigned long id);
+
 #endif
Index: upstream/kernel/cgroup.c
===================================================================
--- upstream.orig/kernel/cgroup.c
+++ upstream/kernel/cgroup.c
@@ -44,6 +44,7 @@
 #include <linux/kmod.h>
 #include <linux/delayacct.h>
 #include <linux/cgroupstats.h>
+#include <linux/file.h>

 #include <asm/atomic.h>

@@ -2442,15 +2443,25 @@ static struct file_operations proc_cgrou
  * At the point that cgroup_fork() is called, 'current' is the parent
  * task, and the passed argument 'child' points to the child task.
  */
-void cgroup_fork(struct task_struct *child)
+void cgroup_fork(struct task_struct *parent, struct task_struct *child)
 {
- task_lock(current);
- child->cgroups = current->cgroups;
+ if (parent == current)
+  task_lock(parent);
+ child->cgroups = parent->cgroups;
  get_css_set(child->cgroups);
- task_unlock(current);
+ if (parent == current)
+  task_unlock(parent);
  INIT_LIST_HEAD(&child->cg_list);
 }

+void cgroup_fork_fromcgroup(struct cgroup *new_cg, struct task_struct *child)
+{
+ mutex_lock(&cgroup_mutex);
+ child->cgroups = find_css_set(child->cgroups, new_cg);
+ INIT_LIST_HEAD(&child->cg_list);
+ mutex_unlock(&cgroup_mutex);
+}
+
 /**
  * cgroup_fork_callbacks - called on a new task very soon before
  * adding it to the tasklist. No need to take any locks since no-one
@@ -2801,3 +2812,117 @@ static void cgroup_release_agent(struct
```

```
  spin_unlock(&release_list_lock);
  mutex_unlock(&cgroup_mutex);
 }
+
+static inline int task_available(struct task_struct *task)
+{
+ if (task == current)
+  return 0;
+ if (task_session(task) == task_session(current))
+  return 0;
+ switch (task->state) {
+ case TASK_RUNNING:
+ case TASK_INTERRUPTIBLE:
+  return 1;
+ default:
+   return 0;
+ }
+}
+
+struct cgroup *cgroup_from_fd(unsigned int fd)
+{
+ struct file *file;
+ struct cgroup *cgroup = NULL;;
+
+ file = fget(fd);
+ if (!file)
+  return NULL;
+
+ if (!file->f_dentry || !file->f_dentry->d_sb)
+  goto out_fput;
+ if (file->f_dentry->d_parent->d_sb->s_magic != CGROUP_SUPER_MAGIC)
+  goto out_fput;
+ if (strcmp(file->f_dentry->d_name.name, "tasks"))
+  goto out_fput;
+
+ cgroup = __d_cgrp(file->f_dentry->d_parent);
+
+out_fput:
+ fput(file);
+ return cgroup;
+}
+
+/*
+ * Takes an integer which is a open fd in current for a valid
+ * cgroupfs file.  Returns a task in that cgroup, with its
+ * refcount bumped.
+ * Since we have an open file on the cgroup tasks file, we
+ * at least don't have to worry about the cgroup being freed
```

```
+ * in the middle of this.
+ */
+struct task_struct *task_from_cgroup_fd(unsigned int fd)
+{
+ struct cgroup *cgroup;
+ struct cgroup_iter it;
+ struct task_struct *task = NULL;
+
+ cgroup = cgroup_from_fd(fd);
+ if (!cgroup)
+  return NULL;
+
+ rcu_read_lock();
+ cgroup_iter_start(cgroup, &it);
+ do {
+  task = cgroup_iter_next(cgroup, &it);
+  if (task)
+   printk(KERN_NOTICE "task %d state %lx\n",
+    task->pid, task->state);
+ } while (task && !task_available(task));
+ cgroup_iter_end(cgroup, &it);
+ if (task)
+  get_task_struct(task);
+ rcu_read_unlock();
+ return task;
+}
+
+/*
+ * is current allowed to hijack tsk?
+ * permission will also be denied elsewhere if
+ *  current may not ptrace tsk
+ *  security_task_alloc(new_task, tsk) returns -EPERM
+ * Here we are only checking whether current may attach
+ * to tsk's cgroup.  If you can't enter the cgroup, you can't
+ * hijack it.
+ *
+ * XXX TODO This means that ns_cgroup.c will need to allow
+ * entering all descendent cgroups, not just the immediate
+ * child.
+ */
+int cgroup_may_hijack(struct task_struct *tsk)
+{
+ int ret = 0;
+ struct cgroupfs_root *root;
+
+ mutex_lock(&cgroup_mutex);
+ for_each_root(root) {
+  struct cgroup_subsys *ss;
```

```
+  struct cgroup *cgroup;
+  int subsys_id;
+
+  /* Skip this hierarchy if it has no active subsystems */
+  if (!root->actual_subsys_bits)
+   continue;
+  get_first_subsys(&root->top_cgroup, NULL, &subsys_id);
+  cgroup = task_cgroup(tsk, subsys_id);
+  for_each_subsys(root, ss) {
+   if (ss->may_hijack) {
+    ret = ss->may_hijack(ss, cgroup, tsk);
+    if (ret)
+     goto out_unlock;
+   }
+  }
+ }
+
+out_unlock:
+ mutex_unlock(&cgroup_mutex);
+ return ret;
+}
Index: upstream/kernel/fork.c
===================================================================
--- upstream.orig/kernel/fork.c
+++ upstream/kernel/fork.c
@@ -189,7 +189,7 @@ static struct task_struct *dup_task_stru
   return NULL;
  }

- setup_thread_stack(tsk, orig);
+ setup_thread_stack(tsk, current);

 #ifdef CONFIG_CC_STACKPROTECTOR
  tsk->stack_canary = get_random_int();
@@ -616,13 +616,14 @@ struct fs_struct *copy_fs_struct(struct

 EXPORT_SYMBOL_GPL(copy_fs_struct);

-static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
+static inline int copy_fs(unsigned long clone_flags,
+  struct task_struct *src, struct task_struct *tsk)
 {
  if (clone_flags & CLONE_FS) {
-  atomic_inc(&current->fs->count);
+  atomic_inc(&src->fs->count);
   return 0;
  }
- tsk->fs = __copy_fs_struct(current->fs);
```

```
+ tsk->fs = __copy_fs_struct(src->fs);
  if (!tsk->fs)
   return -ENOMEM;
  return 0;
@@ -962,6 +963,42 @@ static void rt_mutex_init_task(struct ta
 #endif
 }

+void copy_hijackable_taskinfo(struct task_struct *p,
+  struct task_struct *task)
+{
+ p->uid = task->uid;
+ p->euid = task->euid;
+ p->suid = task->suid;
+ p->fsuid = task->fsuid;
+ p->gid = task->gid;
+ p->egid = task->egid;
+ p->sgid = task->sgid;
+ p->fsgid = task->fsgid;
+ p->cap_effective = task->cap_effective;
+ p->cap_inheritable = task->cap_inheritable;
+ p->cap_permitted = task->cap_permitted;
+ p->keep_capabilities = task->keep_capabilities;
+ p->user = task->user;
+ /*
+  * should keys come from parent or hijack-src?
+  */
+#ifdef CONFIG_SYSVIPC
+ p->sysvsem = task->sysvsem;
+#endif
+ p->fs = task->fs;
+ p->nsproxy = task->nsproxy;
+}
+
+#define HIJACK_SOURCE_TASK 1
+#define HIJACK_SOURCE_CG 2
+struct hijack_source_info {
+ char type;
+ union hijack_source_union {
+  struct task_struct *task;
+  struct cgroup *cgroup;
+ } u;
+};
+
 /*
  * This creates a new process as a copy of the old one,
  * but does not actually start it yet.
@@ -970,7 +1007,8 @@ static void rt_mutex_init_task(struct ta
```

```
  * parts of the process environment (as per the clone
  * flags). The actual kick-off is left to the caller.
  */
-static struct task_struct *copy_process(unsigned long clone_flags,
+static struct task_struct *copy_process(struct hijack_source_info *src,
+     unsigned long clone_flags,
      unsigned long stack_start,
      struct pt_regs *regs,
      unsigned long stack_size,
@@ -980,6 +1018,12 @@ static struct task_struct *copy_process(
  int retval;
  struct task_struct *p;
  int cgroup_callbacks_done = 0;
+ struct task_struct *task;
+
+ if (src->type == HIJACK_SOURCE_TASK)
+  task = src->u.task;
+ else
+  task = current;

  if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
   return ERR_PTR(-EINVAL);
@@ -1007,6 +1051,10 @@ static struct task_struct *copy_process(
  p = dup_task_struct(current);
  if (!p)
   goto fork_out;
+ if (current != task)
+  copy_hijackable_taskinfo(p, task);
+ else if (src->type == HIJACK_SOURCE_CG)
+  copy_hijack_nsproxy(p, src->u.cgroup);

  rt_mutex_init_task(p);

@@ -1084,7 +1132,10 @@ static struct task_struct *copy_process(
 #endif
  p->io_context = NULL;
  p->audit_context = NULL;
- cgroup_fork(p);
+ if (src->type == HIJACK_SOURCE_CG)
+  cgroup_fork_fromcgroup(src->u.cgroup, p);
+ else
+  cgroup_fork(task, p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
   if (IS_ERR(p->mempolicy)) {
@@ -1135,7 +1186,7 @@ static struct task_struct *copy_process(
   goto bad_fork_cleanup_audit;
  if ((retval = copy_files(clone_flags, p)))
```

```
   goto bad_fork_cleanup_semundo;
- if ((retval = copy_fs(clone_flags, p)))
+ if ((retval = copy_fs(clone_flags, task, p)))
   goto bad_fork_cleanup_files;
  if ((retval = copy_sighand(clone_flags, p)))
   goto bad_fork_cleanup_fs;
@@ -1167,7 +1218,7 @@ static struct task_struct *copy_process(
  p->pid = pid_nr(pid);
  p->tgid = p->pid;
  if (clone_flags & CLONE_THREAD)
-  p->tgid = current->tgid;
+  p->tgid = task->tgid;

  p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr : NULL;
  /*
@@ -1378,8 +1429,12 @@ struct task_struct * __cpuinit fork_idle
 {
  struct task_struct *task;
  struct pt_regs regs;
+ struct hijack_source_info src;

- task = copy_process(CLONE_VM, 0, idle_regs(&regs), 0, NULL,
+ src.type = HIJACK_SOURCE_TASK;
+ src.u.task = current;
+
+ task = copy_process(&src, CLONE_VM, 0, idle_regs(&regs), 0, NULL,
     &init_struct_pid);
  if (!IS_ERR(task))
   init_idle(task, cpu);
@@ -1404,29 +1459,43 @@ static int fork_traceflag(unsigned clone
 }

 /*
- *  Ok, this is the main fork-routine.
- *
- * It copies the process, and if successful kick-starts
- * it and waits for it to finish using the VM if required.
+ * if called with task!=current, then caller must ensure that
+ *    1. it has a reference to task
+ *    2. current must have ptrace permission to task
  */
-long do_fork(unsigned long clone_flags,
+long do_fork_task(struct hijack_source_info *src,
+  unsigned long clone_flags,
        unsigned long stack_start,
        struct pt_regs *regs,
        unsigned long stack_size,
        int __user *parent_tidptr,
```

```
        int __user *child_tidptr)
 {
- struct task_struct *p;
+ struct task_struct *p, *task;
  int trace = 0;
  long nr;

+ if (src->type == HIJACK_SOURCE_TASK)
+  task = src->u.task;
+ else
+  task = current;
+ if (task != current) {
+  /* sanity checks */
+  /* we only want to allow hijacking the simplest cases */
+  if (clone_flags & CLONE_SYSVSEM)
+   return -EINVAL;
+  if (current->ptrace)
+   return -EPERM;
+  if (task->ptrace)
+   return -EINVAL;
+ }
  if (unlikely(current->ptrace)) {
   trace = fork_traceflag (clone_flags);
   if (trace)
    clone_flags |= CLONE_PTRACE;
  }

- p = copy_process(clone_flags, stack_start, regs, stack_size,
+ p = copy_process(src, clone_flags, stack_start, regs, stack_size,
    child_tidptr, NULL);
  /*
   * Do this prior waking up the new thread - the thread pointer
@@ -1484,6 +1553,106 @@ long do_fork(unsigned long clone_flags,
  return nr;
 }

+/*
+ *  Ok, this is the main fork-routine.
+ *
+ * It copies the process, and if successful kick-starts
+ * it and waits for it to finish using the VM if required.
+ */
+long do_fork(unsigned long clone_flags,
+       unsigned long stack_start,
+       struct pt_regs *regs,
+       unsigned long stack_size,
+       int __user *parent_tidptr,
+       int __user *child_tidptr)
```

```
+{
+ struct hijack_source_info src = {
+    .type = HIJACK_SOURCE_TASK,
+    .u = { .task = current, },
+ };
+ return do_fork_task(&src, clone_flags, stack_start,
+  regs, stack_size, parent_tidptr, child_tidptr);
+}
+
+/*
+ * Called with task count bumped, drops task count before returning
+ */
+int hijack_task(struct task_struct *task, unsigned long clone_flags,
+  struct pt_regs regs, unsigned long sp)
+{
+ int ret = -EPERM;
+ struct hijack_source_info src = {
+    .type = HIJACK_SOURCE_TASK,
+    .u = { .task = task, },
+ };
+
+ task_lock(task);
+ put_task_struct(task);
+ if (!ptrace_may_attach_locked(task))
+  goto out_unlock_task;
+ if (task == current)
+  goto out_unlock_task;
+ ret = cgroup_may_hijack(task);
+ if (ret)
+  goto out_unlock_task;
+ if (task->ptrace) {
+  ret = -EBUSY;
+  goto out_unlock_task;
+ }
+ ret = do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
+
+out_unlock_task:
+ task_unlock(task);
+ return ret;
+}
+
+int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
+        unsigned long sp)
+{
+ struct task_struct *task;
+
+ rcu_read_lock();
+ task = find_task_by_vpid(pid);
```

```
+ if (task)
+  get_task_struct(task);
+ rcu_read_unlock();
+
+ if (!task)
+  return -EINVAL;
+
+ return hijack_task(task, clone_flags, regs, sp);
+}
+
+int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
+    struct pt_regs regs, unsigned long sp)
+{
+ struct task_struct *task;
+
+ task = task_from_cgroup_fd(fd);
+ if (!task)
+  return -EINVAL;
+
+ return hijack_task(task, clone_flags, regs, sp);
+}
+
+int hijack_ns(unsigned int fd, unsigned long clone_flags,
+    struct pt_regs regs, unsigned long sp)
+{
+ struct hijack_source_info src;
+ struct cgroup *cgroup;
+
+ cgroup = cgroup_from_fd(fd);
+ if (!cgroup)
+  return -EINVAL;
+
+ if (!ns_cgroup_verify(cgroup))
+  return -EINVAL;
+
+ src.type = HIJACK_SOURCE_CG;
+ src.u.cgroup = cgroup;
+ return do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
+}
+
 #ifndef ARCH_MIN_MMSTRUCT_ALIGN
 #define ARCH_MIN_MMSTRUCT_ALIGN 0
 #endif
Index: upstream/kernel/ns_cgroup.c
===================================================================
--- upstream.orig/kernel/ns_cgroup.c
+++ upstream/kernel/ns_cgroup.c
@@ -7,9 +7,11 @@
```

```
 #include <linux/module.h>
 #include <linux/cgroup.h>
 #include <linux/fs.h>
+#include <linux/nsproxy.h>

 struct ns_cgroup {
  struct cgroup_subsys_state css;
+ struct nsproxy *nsproxy;
  spinlock_t lock;
 };

@@ -22,9 +24,51 @@ static inline struct ns_cgroup *cgroup_t
        struct ns_cgroup, css);
 }

-int ns_cgroup_clone(struct task_struct *task)
+int ns_cgroup_clone(struct task_struct *task, struct nsproxy *nsproxy)
 {
- return cgroup_clone(task, &ns_subsys);
+ struct cgroup *cgroup;
+ struct ns_cgroup *ns_cgroup;
+ int ret = cgroup_clone(task, &ns_subsys);
+
+ if (ret)
+  return ret;
+
+ cgroup = task_cgroup(task, ns_subsys_id);
+ ns_cgroup = cgroup_to_ns(cgroup);
+ ns_cgroup->nsproxy = nsproxy;
+ get_nsproxy(nsproxy);
+
+ return 0;
+}
+
+int ns_cgroup_verify(struct cgroup *cgroup)
+{
+ struct cgroup_subsys_state *css;
+ struct ns_cgroup *ns_cgroup;
+
+ css = cgroup_subsys_state(cgroup, ns_subsys_id);
+ if (!css)
+  return 0;
+ ns_cgroup = container_of(css, struct ns_cgroup, css);
+ if (!ns_cgroup->nsproxy)
+  return 0;
+ return 1;
+}
+
```

```
+/*
+ * this shouldn't be called unless ns_cgroup_verify() has
+ * confirmed that there is a ns_cgroup in this cgroup
+ *
+ * tsk is not yet running, and has not yet taken a reference
+ * to it's previous ->nsproxy, so we just do a simple assignment
+ * rather than switch_task_namespaces()
+ */
+void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup)
+{
+ struct ns_cgroup *ns_cgroup;
+
+ ns_cgroup = cgroup_to_ns(cgroup);
+ tsk->nsproxy = ns_cgroup->nsproxy;
 }

 /*
@@ -60,6 +104,42 @@ static int ns_can_attach(struct cgroup_s
  return 0;
 }

+static void ns_attach(struct cgroup_subsys *ss,
+      struct cgroup *cgroup, struct cgroup *oldcgroup,
+      struct task_struct *tsk)
+{
+ struct ns_cgroup *ns_cgroup = cgroup_to_ns(cgroup);
+
+ if (likely(ns_cgroup->nsproxy))
+  return;
+
+ spin_lock(&ns_cgroup->lock);
+ if (!ns_cgroup->nsproxy) {
+  ns_cgroup->nsproxy = tsk->nsproxy;
+  get_nsproxy(ns_cgroup->nsproxy);
+ }
+ spin_unlock(&ns_cgroup->lock);
+}
+
+/*
+ * only allow hijacking child namespaces
+ * Q: is it crucial to prevent hijacking a task in your same cgroup?
+ */
+static int ns_may_hijack(struct cgroup_subsys *ss,
+  struct cgroup *new_cgroup, struct task_struct *task)
+{
+ if (current == task)
+  return -EINVAL;
+
```

```
+ if (!capable(CAP_SYS_ADMIN))
+  return -EPERM;
+
+ if (!cgroup_is_descendant(new_cgroup))
+  return -EPERM;
+
+ return 0;
+}
+
 /*
  * Rules: you can only create a cgroup if
  *     1. you are capable(CAP_SYS_ADMIN)
@@ -88,12 +168,16 @@ static void ns_destroy(struct cgroup_sub
  struct ns_cgroup *ns_cgroup;

  ns_cgroup = cgroup_to_ns(cgroup);
+ if (ns_cgroup->nsproxy)
+  put_nsproxy(ns_cgroup->nsproxy);
  kfree(ns_cgroup);
 }

 struct cgroup_subsys ns_subsys = {
  .name = "ns",
  .can_attach = ns_can_attach,
+ .attach = ns_attach,
+ .may_hijack = ns_may_hijack,
  .create = ns_create,
  .destroy  = ns_destroy,
  .subsys_id = ns_subsys_id,
Index: upstream/kernel/nsproxy.c
===================================================================
--- upstream.orig/kernel/nsproxy.c
+++ upstream/kernel/nsproxy.c
@@ -144,7 +144,7 @@ int copy_namespaces(unsigned long flags,
   goto out;
  }

- err = ns_cgroup_clone(tsk);
+ err = ns_cgroup_clone(tsk, new_ns);
  if (err) {
   put_nsproxy(new_ns);
   goto out;
@@ -196,7 +196,7 @@ int unshare_nsproxy_namespaces(unsigned
   goto out;
  }

- err = ns_cgroup_clone(current);
+ err = ns_cgroup_clone(current, *new_nsp);
```

```
 if (err)
  put_nsproxy(*new_nsp);

Index: upstream/kernel/ptrace.c
===================================================================
--- upstream.orig/kernel/ptrace.c
+++ upstream/kernel/ptrace.c
@@ -159,6 +159,13 @@ int ptrace_may_attach(struct task_struct
  return !err;
 }

+int ptrace_may_attach_locked(struct task_struct *task)
+{
+ int err;
+ err = may_attach(task);
+ return !err;
+}
+
 int ptrace_attach(struct task_struct *task)
 {
  int retval;
```

_____

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

---

Subject: [PATCH 2/2] hijack: update task_alloc_security
Posted by Mark Nelson on Tue, 27 Nov 2007 02:00:49 GMT
View Forum Message <> Reply to Message

Subject: [PATCH 2/2] hijack: update task_alloc_security

Update task_alloc_security() to take the hijacked task as a second
argument.

For the selinux version, refuse permission if hijack_src!=current,
since we have no idea what the proper behavior is. Even if we
assume that the resulting child should be in the hijacked task's
domain, depending on the policy that may not be enough information
since init_t executing /bin/bash could result in a different domain
than login_t executing /bin/bash.

Signed-off-by: Serge Hallyn <serue@us.ibm.com>
Signed-off-by: Mark Nelson <markn@au1.ibm.com>
---
 include/linux/security.h | 12 ++++++++---
 kernel/fork.c | 2 +-

```
 security/dummy.c         |    3 ++-
 security/security.c      |    4 ++--
 security/selinux/hooks.c |    6 +++++-
 5 files changed, 19 insertions(+), 8 deletions(-)

Index: upstream/include/linux/security.h
===================================================================
--- upstream.orig/include/linux/security.h
+++ upstream/include/linux/security.h
@@ -545,9 +545,13 @@ struct request_sock;
  * Return 0 if permission is granted.
  * @task_alloc_security:
  * @p contains the task_struct for child process.
+ * @task contains the task_struct for process to be hijacked
  * Allocate and attach a security structure to the p->security field. The
  * security field is initialized to NULL when the task structure is
  * allocated.
+ * @task will usually be current.  If it is not equal to current, then
+ * a sys_hijack system call is going on, and current is asking for a
+ * child to be created in the context of the hijack src, @task.
  * Return 0 if operation was successful.
  * @task_free_security:
  * @p contains the task_struct for process.
@@ -1301,7 +1305,8 @@ struct security_operations {
  int (*dentry_open)  (struct file *file);

  int (*task_create) (unsigned long clone_flags);
- int (*task_alloc_security) (struct task_struct * p);
+ int (*task_alloc_security) (struct task_struct *p,
+         struct task_struct *task);
  void (*task_free_security) (struct task_struct * p);
  int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
  int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
@@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
 int security_file_receive(struct file *file);
 int security_dentry_open(struct file *file);
 int security_task_create(unsigned long clone_flags);
-int security_task_alloc(struct task_struct *p);
+int security_task_alloc(struct task_struct *p, struct task_struct *task);
 void security_task_free(struct task_struct *p);
 int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int flags);
 int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
@@ -2021,7 +2026,8 @@ static inline int security_task_create (
  return 0;
 }

-static inline int security_task_alloc (struct task_struct *p)
+static inline int security_task_alloc(struct task_struct *p,
```

```
+          struct task_struct *task)
 {
  return 0;
 }
Index: upstream/kernel/fork.c
===================================================================
--- upstream.orig/kernel/fork.c
+++ upstream/kernel/fork.c
@@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
  /* Perform scheduler related setup. Assign this task to a CPU. */
  sched_fork(p, clone_flags);

- if ((retval = security_task_alloc(p)))
+ if ((retval = security_task_alloc(p, task)))
    goto bad_fork_cleanup_policy;
  if ((retval = audit_alloc(p)))
    goto bad_fork_cleanup_security;
Index: upstream/security/dummy.c
===================================================================
--- upstream.orig/security/dummy.c
+++ upstream/security/dummy.c
@@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
  return 0;
 }

-static int dummy_task_alloc_security (struct task_struct *p)
+static int dummy_task_alloc_security(struct task_struct *p,
+          struct task_struct *task)
 {
  return 0;
 }
Index: upstream/security/security.c
===================================================================
--- upstream.orig/security/security.c
+++ upstream/security/security.c
@@ -568,9 +568,9 @@ int security_task_create(unsigned long c
  return security_ops->task_create(clone_flags);
 }

-int security_task_alloc(struct task_struct *p)
+int security_task_alloc(struct task_struct *p, struct task_struct *task)
 {
- return security_ops->task_alloc_security(p);
+ return security_ops->task_alloc_security(p, task);
 }

 void security_task_free(struct task_struct *p)
Index: upstream/security/selinux/hooks.c
```

```
===================================================================
--- upstream.orig/security/selinux/hooks.c
+++ upstream/security/selinux/hooks.c
@@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
  return task_has_perm(current, current, PROCESS__FORK);
 }

-static int selinux_task_alloc_security(struct task_struct *tsk)
+static int selinux_task_alloc_security(struct task_struct *tsk,
+           struct task_struct *hijack_src)
 {
  struct task_security_struct *tsec1, *tsec2;
  int rc;

+ if (hijack_src != current)
+  return -EPERM;
+
  tsec1 = current->security;

  rc = task_alloc_security(tsk);
```

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Casey Schaufler on Tue, 27 Nov 2007 05:04:37 GMT
View Forum Message <> Reply to Message

--- Mark Nelson <markn@au1.ibm.com> wrote:

> Subject: [PATCH 2/2] hijack: update task_alloc_security
>
> Update task_alloc_security() to take the hijacked task as a second
> argument.

Could y'all bring me up to speed on what this is intended to
accomplish so that I can understand the Smack implications?

Thank you.

> For the selinux version, refuse permission if hijack_src!=current,
> since we have no idea what the proper behavior is. Even if we
> assume that the resulting child should be in the hijacked task's
> domain, depending on the policy that may not be enough information
> since init_t executing /bin/bash could result in a different domain
> than login_t executing /bin/bash.

>
> Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> ---
> include/linux/security.h | 12 +++++++++---
> kernel/fork.c | 2 +-
> security/dummy.c | 3 ++-
> security/security.c | 4 ++--
> security/selinux/hooks.c | 6 +++++-
> 5 files changed, 19 insertions(+), 8 deletions(-)
>
> Index: upstream/include/linux/security.h
> ===
> --- upstream.orig/include/linux/security.h
> +++ upstream/include/linux/security.h
> @@ -545,9 +545,13 @@ struct request_sock;
>   * Return 0 if permission is granted.
>   * @task_alloc_security:
>   * @p contains the task_struct for child process.
> + * @task contains the task_struct for process to be hijacked
>   * Allocate and attach a security structure to the p->security field. The
>   * security field is initialized to NULL when the task structure is
>   * allocated.
> + * @task will usually be current. If it is not equal to current, then
> + * a sys_hijack system call is going on, and current is asking for a
> + * child to be created in the context of the hijack src, @task.
>   * Return 0 if operation was successful.
>   * @task_free_security:
>   * @p contains the task_struct for process.
> @@ -1301,7 +1305,8 @@ struct security_operations {
> 	int (*dentry_open) (struct file *file);
>
> 	int (*task_create) (unsigned long clone_flags);
> -	int (*task_alloc_security) (struct task_struct * p);
> +	int (*task_alloc_security) (struct task_struct *p,
> +	 struct task_struct *task);
> 	void (*task_free_security) (struct task_struct * p);
> 	int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> 	int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> int security_file_receive(struct file *file);
> int security_dentry_open(struct file *file);
> int security_task_create(unsigned long clone_flags);
> -int security_task_alloc(struct task_struct *p);
> +int security_task_alloc(struct task_struct *p, struct task_struct *task);
> void security_task_free(struct task_struct *p);
> int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int flags);
> int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,

```
> @@ -2021,7 +2026,8 @@ static inline int security_task_create (
>   return 0;
>  }
>
> -static inline int security_task_alloc (struct task_struct *p)
> +static inline int security_task_alloc(struct task_struct *p,
> +           struct task_struct *task)
>  {
>   return 0;
>  }
> Index: upstream/kernel/fork.c
> ===================================================================
> --- upstream.orig/kernel/fork.c
> +++ upstream/kernel/fork.c
> @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
>   /* Perform scheduler related setup. Assign this task to a CPU. */
>   sched_fork(p, clone_flags);
>
> - if ((retval = security_task_alloc(p)))
> + if ((retval = security_task_alloc(p, task)))
>    goto bad_fork_cleanup_policy;
>   if ((retval = audit_alloc(p)))
>    goto bad_fork_cleanup_security;
> Index: upstream/security/dummy.c
> ===================================================================
> --- upstream.orig/security/dummy.c
> +++ upstream/security/dummy.c
> @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
>   return 0;
>  }
>
> -static int dummy_task_alloc_security (struct task_struct *p)
> +static int dummy_task_alloc_security(struct task_struct *p,
> +           struct task_struct *task)
>  {
>   return 0;
>  }
> Index: upstream/security/security.c
> ===================================================================
> --- upstream.orig/security/security.c
> +++ upstream/security/security.c
> @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
>   return security_ops->task_create(clone_flags);
>  }
>
> -int security_task_alloc(struct task_struct *p)
> +int security_task_alloc(struct task_struct *p, struct task_struct *task)
>  {
```

> - return security_ops->task_alloc_security(p);
> + return security_ops->task_alloc_security(p, task);
> }
>
> void security_task_free(struct task_struct *p)
> Index: upstream/security/selinux/hooks.c
> ===
> --- upstream.orig/security/selinux/hooks.c
> +++ upstream/security/selinux/hooks.c
> @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
>   return task_has_perm(current, current, PROCESS__FORK);
> }
>
> -static int selinux_task_alloc_security(struct task_struct *tsk)
> +static int selinux_task_alloc_security(struct task_struct *tsk,
> +   struct task_struct *hijack_src)
> {
>   struct task_security_struct *tsec1, *tsec2;
>   int rc;
>
> + if (hijack_src != current)
> + return -EPERM;
> +
>   tsec1 = current->security;
>
>   rc = task_alloc_security(tsk);
> -
> To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
>
>

Casey Schaufler
casey@schaufler-ca.com

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Joshua Brindle on Tue, 27 Nov 2007 05:52:21 GMT
View Forum Message <> Reply to Message

Mark Nelson wrote:
> Subject: [PATCH 2/2] hijack: update task_alloc_security
>
> Update task_alloc_security() to take the hijacked task as a second
> argument.
>
> For the selinux version, refuse permission if hijack_src!=current,
> since we have no idea what the proper behavior is. Even if we
> assume that the resulting child should be in the hijacked task's
> domain, depending on the policy that may not be enough information
> since init_t executing /bin/bash could result in a different domain
> than login_t executing /bin/bash.
>
>
This means its basically not possible to hijack tasks with SELinux
right? It would be a shame if this weren't useful to people running SELinux.

It seems to me (I may be wrong, I'm sure someone will let me know if I
am) that the right way to handle this with SELinux is to check to see if
the current task (caller of sys_hijack) has permission to ptrace (or
some other permission deemed suitable, perhaps a new one) and if so copy
the security blob pointer from the hijacked task to the new one (we
don't want tranquility problems).

 From your paragraph above it seems like you were thinking there should
be a transition at hijack time but we don't automatically transition
anywhere except exec.

Anyway, I just don't think you should completely disable this for
SELinux users.

> Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> ---
> include/linux/security.h | 12 ++++++++---
> kernel/fork.c | 2 +-
> security/dummy.c | 3 ++-
> security/security.c | 4 ++--
> security/selinux/hooks.c | 6 +++++-
> 5 files changed, 19 insertions(+), 8 deletions(-)
>
> Index: upstream/include/linux/security.h
> ===
> --- upstream.orig/include/linux/security.h
> +++ upstream/include/linux/security.h
> @@ -545,9 +545,13 @@ struct request_sock;
>   * Return 0 if permission is granted.
>   * @task_alloc_security:

```
>   * @p contains the task_struct for child process.
> + * @task contains the task_struct for process to be hijacked
>   * Allocate and attach a security structure to the p->security field. The
>   * security field is initialized to NULL when the task structure is
>   * allocated.
> + * @task will usually be current.  If it is not equal to current, then
> + * a sys_hijack system call is going on, and current is asking for a
> + * child to be created in the context of the hijack src, @task.
>   * Return 0 if operation was successful.
>   * @task_free_security:
>   * @p contains the task_struct for process.
> @@ -1301,7 +1305,8 @@ struct security_operations {
>   int (*dentry_open)  (struct file *file);
>
>   int (*task_create) (unsigned long clone_flags);
> - int (*task_alloc_security) (struct task_struct * p);
> + int (*task_alloc_security) (struct task_struct *p,
> +         struct task_struct *task);
>   void (*task_free_security) (struct task_struct * p);
>   int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
>   int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
>  int security_file_receive(struct file *file);
>  int security_dentry_open(struct file *file);
>  int security_task_create(unsigned long clone_flags);
> -int security_task_alloc(struct task_struct *p);
> +int security_task_alloc(struct task_struct *p, struct task_struct *task);
>  void security_task_free(struct task_struct *p);
>  int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int flags);
>  int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> @@ -2021,7 +2026,8 @@ static inline int security_task_create (
>   return 0;
>  }
>
> -static inline int security_task_alloc (struct task_struct *p)
> +static inline int security_task_alloc(struct task_struct *p,
> +           struct task_struct *task)
>  {
>   return 0;
>  }
> Index: upstream/kernel/fork.c
> ===================================================================
> --- upstream.orig/kernel/fork.c
> +++ upstream/kernel/fork.c
> @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
>   /* Perform scheduler related setup. Assign this task to a CPU. */
>   sched_fork(p, clone_flags);
>
```

> - if ((retval = security_task_alloc(p)))
> + if ((retval = security_task_alloc(p, task)))
>   goto bad_fork_cleanup_policy;
>   if ((retval = audit_alloc(p)))
>   goto bad_fork_cleanup_security;
> Index: upstream/security/dummy.c
> ===
> --- upstream.orig/security/dummy.c
> +++ upstream/security/dummy.c
> @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
>   return 0;
> }
>
> -static int dummy_task_alloc_security (struct task_struct *p)
> +static int dummy_task_alloc_security(struct task_struct *p,
> +   struct task_struct *task)
> {
>   return 0;
> }
> Index: upstream/security/security.c
> ===
> --- upstream.orig/security/security.c
> +++ upstream/security/security.c
> @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
>   return security_ops->task_create(clone_flags);
> }
>
> -int security_task_alloc(struct task_struct *p)
> +int security_task_alloc(struct task_struct *p, struct task_struct *task)
> {
> - return security_ops->task_alloc_security(p);
> + return security_ops->task_alloc_security(p, task);
> }
>
> void security_task_free(struct task_struct *p)
> Index: upstream/security/selinux/hooks.c
> ===
> --- upstream.orig/security/selinux/hooks.c
> +++ upstream/security/selinux/hooks.c
> @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
>   return task_has_perm(current, current, PROCESS__FORK);
> }
>
> -static int selinux_task_alloc_security(struct task_struct *tsk)
> +static int selinux_task_alloc_security(struct task_struct *tsk,
> +   struct task_struct *hijack_src)
> {
>   struct task_security_struct *tsec1, *tsec2;

>   int rc;
>
> + if (hijack_src != current)
> + return -EPERM;
> +
>   tsec1 = current->security;
>
>   rc = task_alloc_security(tsk);
> -
> To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
>

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Crispin Cowan on Tue, 27 Nov 2007 06:58:31 GMT
View Forum Message <> Reply to Message

Just the name "sys_hijack" makes me concerned.

This post describes a bunch of "what", but doesn't tell us about "why"
we would want this. What is it for?

And I second Casey's concern about careful management of the privilege
required to "hijack" a process.

Crispin

Mark Nelson wrote:
> Here's the latest version of sys_hijack.
> Apologies for its lateness.
>
> Thanks!
>
> Mark.
>
> Subject: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
>
> Move most of do_fork() into a new do_fork_task() which acts on
> a new argument, task, rather than on current. do_fork() becomes

> a call to do_fork_task(current, ...).
>
> Introduce sys_hijack (for i386 and s390 only so far). It is like
> clone, but in place of a stack pointer (which is assumed null) it
> accepts a pid. The process identified by that pid is the one
> which is actually cloned. Some state - including the file
> table, the signals and sighand (and hence tty), and the ->parent
> are taken from the calling process.
>
> A process to be hijacked may be identified by process id, in the
> case of HIJACK_PID. Alternatively, in the case of HIJACK_CG an
> open fd for a cgroup 'tasks' file may be specified. The first
> available task in that cgroup will then be hijacked.
>
> HIJACK_NS is implemented as a third hijack method. The main
> purpose is to allow entering an empty cgroup without having
> to keep a task alive in the target cgroup. When HIJACK_NS
> is called, only the cgroup and nsproxy are copied from the
> cgroup. Security, user, and rootfs info is not retained
> in the cgroups and so cannot be copied to the child task.
>
> In order to hijack a process, the calling process must be
> allowed to ptrace the target.
>
> Sending sigstop to the hijacked task can trick its parent shell
> (if it is a shell foreground task) into thinking it should retake
> its tty.
>
> So try not sending SIGSTOP, and instead hold the task_lock over
> the hijacked task throughout the do_fork_task() operation.
> This is really dangerous. I've fixed cgroup_fork() to not
> task_lock(task) in the hijack case, but there may well be other
> code called during fork which can under "some circumstances"
> task_lock(task).
>
> Still, this is working for me.
>
> The effect is a sort of namespace enter. The following program
> uses sys_hijack to 'enter' all namespaces of the specified task.
> For instance in one terminal, do
>
> mount -t cgroup -ons cgroup /cgroup
> hostname
>   qemu
> ns_exec -u /bin/sh
>   hostname serge
>   echo $$
>   1073

>   cat /proc/$$/cgroup
>   ns:/node_1073
>
> In another terminal then do
>
> hostname
>   qemu
> cat /proc/$$/cgroup
>   ns:/
> hijack pid 1073
>   hostname
>   serge
>   cat /proc/$$/cgroup
>   ns:/node_1073
> hijack cgroup /cgroup/node_1073/tasks
>
> Changelog:
> Aug 23: send a stop signal to the hijacked process
>   (like ptrace does).
> Oct 09: Update for 2.6.23-rc8-mm2 (mainly pidns)
>   Don't take task_lock under rcu_read_lock
>   Send hijacked process to cgroup_fork() as
>   the first argument.
>   Removed some unneeded task_locks.
> Oct 16: Fix bug introduced into alloc_pid.
> Oct 16: Add 'int which' argument to sys_hijack to
>   allow later expansion to use cgroup in place
>   of pid to specify what to hijack.
> Oct 24: Implement hijack by open cgroup file.
> Nov 02: Switch copying of task info: do full copy
>   from current, then copy relevant pieces from
>   hijacked task.
> Nov 06: Verbatim task_struct copy now comes from current,
>   after which copy_hijackable_taskinfo() copies
>   relevant context pieces from the hijack source.
> Nov 07: Move arch-independent hijack code to kernel/fork.c
> Nov 07: powerpc and x86_64 support (Mark Nelson)
> Nov 07: Don't allow hijacking members of same session.
> Nov 07: introduce cgroup_may_hijack, and may_hijack hook to
>   cgroup subsystems. The ns subsystem uses this to
>   enforce the rule that one may only hijack descendent
>   namespaces.
> Nov 07: s390 support
> Nov 08: don't send SIGSTOP to hijack source task
> Nov 10: cache reference to nsproxy in ns cgroup for use in
>   hijacking an empty cgroup.
> Nov 10: allow partial hijack of empty cgroup
> Nov 13: don't double-get cgroup for hijack_ns

> find_css_set() actually returns the set with a
> reference already held, so cgroup_fork_fromcgroup()
> by doing a get_css_set() was getting a second
> reference. Therefore after exiting the hijack
> task we could not rmdir the csgroup.
> Nov 22: temporarily remove x86_64 and powerpc support
> Nov 27: rebased on 2.6.24-rc3
>
> ==
> hijack.c
> ==
> /*
> * Your options are:
> * hijack pid 1078
> * hijack cgroup /cgroup/node_1078/tasks
> * hijack ns /cgroup/node_1078/tasks
> */
>
> #define _BSD_SOURCE
> #include <unistd.h>
> #include <sys/syscall.h>
> #include <sys/types.h>
> #include <sys/wait.h>
> #include <sys/stat.h>
> #include <fcntl.h>
> #include <sched.h>
> #include <stdio.h>
> #include <stdlib.h>
> #include <string.h>
>
> #if __i386__
> #   define __NR_hijack 325
> #elif __s390x__
> #   define __NR_hijack 319
> #else
> #   error "Architecture not supported"
> #endif
>
> #ifndef CLONE_NEWUTS
> #define CLONE_NEWUTS 0x04000000
> #endif
>
> void usage(char *me)
> {
> printf("Usage: %s pid <pid>\n", me);
> printf(" | %s cgroup <cgroup_tasks_file>\n", me);
> printf(" | %s ns <cgroup_tasks_file>\n", me);
> exit(1);

```
> }
>
> int exec_shell(void)
> {
>  execl("/bin/sh", "/bin/sh", NULL);
> }
>
> #define HIJACK_PID 1
> #define HIJACK_CG 2
> #define HIJACK_NS 3
>
> int main(int argc, char *argv[])
> {
>  int id;
>  int ret;
>  int status;
>  int which_hijack;
>
>  if (argc < 3 || !strcmp(argv[1], "-h"))
>   usage(argv[0]);
>  if (strcmp(argv[1], "cgroup") == 0)
>   which_hijack = HIJACK_CG;
>  else if (strcmp(argv[1], "ns") == 0)
>   which_hijack = HIJACK_NS;
>  else
>   which_hijack = HIJACK_PID;
>
>  switch(which_hijack) {
>   case HIJACK_PID:
>    id = atoi(argv[2]);
>    printf("hijacking pid %d\n", id);
>    break;
>   case HIJACK_CG:
>   case HIJACK_NS:
>    id = open(argv[2], O_RDONLY);
>    if (id == -1) {
>     perror("cgroup open");
>     return 1;
>    }
>    break;
>  }
>
>  ret = syscall(__NR_hijack, SIGCHLD, which_hijack, (unsigned long)id);
>
>  if (which_hijack != HIJACK_PID)
>   close(id);
>  if  (ret == 0) {
>   return exec_shell();
```

> } else if (ret < 0) {
>   perror("sys_hijack");
> } else {
>   printf("waiting on cloned process %d\n", ret);
>   while(waitpid(-1, &status, __WALL) != -1)
>   ;
>   printf("cloned process exited with %d (waitpid ret %d)\n",
>   status, ret);
> }
>
> return ret;
> }
> ==
>
> Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> ---
> Documentation/cgroups.txt | 9 +
> arch/s390/kernel/process.c | 21 +++
> arch/x86/kernel/process_32.c | 24 ++++
> arch/x86/kernel/syscall_table_32.S | 1
> include/asm-x86/unistd_32.h | 3
> include/linux/cgroup.h | 28 ++++-
> include/linux/nsproxy.h | 12 +-
> include/linux/ptrace.h | 1
> include/linux/sched.h | 19 +++
> include/linux/syscalls.h | 2
> kernel/cgroup.c | 133 ++++++++++++++++++++++-
> kernel/fork.c | 201 +++++++++++++++++++++++++++++++++---
> kernel/ns_cgroup.c | 88 +++++++++++++++-
> kernel/nsproxy.c | 4
> kernel/ptrace.c | 7 +
> 15 files changed, 523 insertions(+), 30 deletions(-)
>
> Index: upstream/arch/s390/kernel/process.c
> ==
> --- upstream.orig/arch/s390/kernel/process.c
> +++ upstream/arch/s390/kernel/process.c
> @@ -321,6 +321,27 @@ asmlinkage long sys_clone(void)
>   parent_tidptr, child_tidptr);
> }
>
> +asmlinkage long sys_hijack(void)
> +{
> + struct pt_regs *regs = task_pt_regs(current);
> + unsigned long sp = regs->orig_gpr2;
> + unsigned long clone_flags = regs->gprs[3];
> + int which = regs->gprs[4];

> + unsigned int fd;
> + pid_t pid;
> +
> + switch (which) {
> + case HIJACK_PID:
> + pid = regs->gprs[5];
> + return hijack_pid(pid, clone_flags, *regs, sp);
> + case HIJACK_CGROUP:
> + fd = (unsigned int) regs->gprs[5];
> + return hijack_cgroup(fd, clone_flags, *regs, sp);
> + default:
> + return -EINVAL;
> + }
> +}
> +
> /*
>   * This is trivial, and on the face of it looks like it
>   * could equally well be done in user mode.
> Index: upstream/arch/x86/kernel/process_32.c
> ===
> --- upstream.orig/arch/x86/kernel/process_32.c
> +++ upstream/arch/x86/kernel/process_32.c
> @@ -37,6 +37,7 @@
> #include <linux/personality.h>
> #include <linux/tick.h>
> #include <linux/percpu.h>
> +#include <linux/cgroup.h>
>
> #include <asm/uaccess.h>
> #include <asm/pgtable.h>
> @@ -781,6 +782,29 @@ asmlinkage int sys_clone(struct pt_regs
>   return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
> }
>
> +asmlinkage int sys_hijack(struct pt_regs regs)
> +{
> + unsigned long sp = regs.esp;
> + unsigned long clone_flags = regs.ebx;
> + int which = regs.ecx;
> + unsigned int fd;
> + pid_t pid;
> +
> + switch (which) {
> + case HIJACK_PID:
> + pid = regs.edx;
> + return hijack_pid(pid, clone_flags, regs, sp);
> + case HIJACK_CGROUP:
> + fd = (unsigned int) regs.edx;

> + return hijack_cgroup(fd, clone_flags, regs, sp);
> + case HIJACK_NS:
> + fd = (unsigned int) regs.edx;
> + return hijack_ns(fd, clone_flags, regs, sp);
> + default:
> + return -EINVAL;
> + }
> +}
> +
> /*
>   * This is trivial, and on the face of it looks like it
>   * could equally well be done in user mode.
> Index: upstream/arch/x86/kernel/syscall_table_32.S
> ===
> --- upstream.orig/arch/x86/kernel/syscall_table_32.S
> +++ upstream/arch/x86/kernel/syscall_table_32.S
> @@ -324,3 +324,4 @@ ENTRY(sys_call_table)
>   .long sys_timerfd
>   .long sys_eventfd
>   .long sys_fallocate
> + .long sys_hijack /* 325 */
> Index: upstream/Documentation/cgroups.txt
> ===
> --- upstream.orig/Documentation/cgroups.txt
> +++ upstream/Documentation/cgroups.txt
> @@ -495,6 +495,15 @@ LL=cgroup_mutex
> Called after the task has been attached to the cgroup, to allow any
> post-attachment activity that requires memory allocations or blocking.
>
> +int may_hijack(struct cgroup_subsys *ss, struct cgroup *cont,
> +   struct task_struct *task)
> +LL=cgroup_mutex
> +
> +Called prior to hijacking a task. Current is cloning a new child
> +which is hijacking cgroup, namespace, and security context from
> +the target task. Called with the hijacked task locked. Return
> +0 to allow.
> +
> void fork(struct cgroup_subsy *ss, struct task_struct *task)
> LL=callback_mutex, maybe read_lock(tasklist_lock)
>
> Index: upstream/include/asm-x86/unistd_32.h
> ===
> --- upstream.orig/include/asm-x86/unistd_32.h
> +++ upstream/include/asm-x86/unistd_32.h
> @@ -330,10 +330,11 @@
> #define __NR_timerfd 322
> #define __NR_eventfd 323

```
>  #define __NR_fallocate  324
> +#define __NR_hijack  325
>
>  #ifdef __KERNEL__
>
> -#define NR_syscalls 325
> +#define NR_syscalls 326
>
>  #define __ARCH_WANT_IPC_PARSE_VERSION
>  #define __ARCH_WANT_OLD_READDIR
> Index: upstream/include/linux/cgroup.h
> ===================================================================
> --- upstream.orig/include/linux/cgroup.h
> +++ upstream/include/linux/cgroup.h
> @@ -14,19 +14,23 @@
>  #include <linux/nodemask.h>
>  #include <linux/rcupdate.h>
>  #include <linux/cgroupstats.h>
> +#include <linux/err.h>
>
>  #ifdef CONFIG_CGROUPS
>
>  struct cgroupfs_root;
>  struct cgroup_subsys;
>  struct inode;
> +struct cgroup;
>
>  extern int cgroup_init_early(void);
>  extern int cgroup_init(void);
>  extern void cgroup_init_smp(void);
>  extern void cgroup_lock(void);
>  extern void cgroup_unlock(void);
> -extern void cgroup_fork(struct task_struct *p);
> +extern void cgroup_fork(struct task_struct *parent, struct task_struct *p);
> +extern void cgroup_fork_fromcgroup(struct cgroup *new_cg,
> +     struct task_struct *child);
>  extern void cgroup_fork_callbacks(struct task_struct *p);
>  extern void cgroup_post_fork(struct task_struct *p);
>  extern void cgroup_exit(struct task_struct *p, int run_callbacks);
> @@ -236,6 +240,8 @@ struct cgroup_subsys {
>   void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cont);
>   int (*can_attach)(struct cgroup_subsys *ss,
>       struct cgroup *cont, struct task_struct *tsk);
> + int (*may_hijack)(struct cgroup_subsys *ss,
> +     struct cgroup *cont, struct task_struct *tsk);
>   void (*attach)(struct cgroup_subsys *ss, struct cgroup *cont,
>     struct cgroup *old_cont, struct task_struct *tsk);
>   void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
```

> @@ -304,12 +310,21 @@ struct task_struct *cgroup_iter_next(str
> struct cgroup_iter *it);
> void cgroup_iter_end(struct cgroup *cont, struct cgroup_iter *it);
>
> +struct cgroup *cgroup_from_fd(unsigned int fd);
> +struct task_struct *task_from_cgroup_fd(unsigned int fd);
> +int cgroup_may_hijack(struct task_struct *tsk);
> #else /* !CONFIG_CGROUPS */
> +struct cgroup {
> +};
>
> static inline int cgroup_init_early(void) { return 0; }
> static inline int cgroup_init(void) { return 0; }
> static inline void cgroup_init_smp(void) {}
> -static inline void cgroup_fork(struct task_struct *p) {}
> +static inline void cgroup_fork(struct task_struct *parent,
> + struct task_struct *p) {}
> +static inline void cgroup_fork_fromcgroup(struct cgroup *new_cg,
> + struct task_struct *child) {}
> +
> static inline void cgroup_fork_callbacks(struct task_struct *p) {}
> static inline void cgroup_post_fork(struct task_struct *p) {}
> static inline void cgroup_exit(struct task_struct *p, int callbacks) {}
> @@ -322,6 +337,15 @@ static inline int cgroupstats_build(stru
> return -EINVAL;
> }
>
> +static inline struct cgroup *cgroup_from_fd(unsigned int fd) { return NULL; }
> +static inline struct task_struct *task_from_cgroup_fd(unsigned int fd)
> +{
> + return ERR_PTR(-EINVAL);
> +}
> +static inline int cgroup_may_hijack(struct task_struct *tsk)
> +{
> + return 0;
> +}
> #endif /* !CONFIG_CGROUPS */
>
> #endif /* _LINUX_CGROUP_H */
> Index: upstream/include/linux/nsproxy.h
> ===
> --- upstream.orig/include/linux/nsproxy.h
> +++ upstream/include/linux/nsproxy.h
> @@ -3,6 +3,7 @@
>
> #include <linux/spinlock.h>
> #include <linux/sched.h>
> +#include <linux/err.h>

>
> struct mnt_namespace;
> struct uts_namespace;
> @@ -81,10 +82,17 @@ static inline void get_nsproxy(struct ns
>   atomic_inc(&ns->count);
> }
>
> +struct cgroup;
> #ifdef CONFIG_CGROUP_NS
> -int ns_cgroup_clone(struct task_struct *tsk);
> +int ns_cgroup_clone(struct task_struct *tsk, struct nsproxy *nsproxy);
> +int ns_cgroup_verify(struct cgroup *cgroup);
> +void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup);
> #else
> -static inline int ns_cgroup_clone(struct task_struct *tsk) { return 0; }
> +static inline int ns_cgroup_clone(struct task_struct *tsk,
> + struct nsproxy *nsproxy) { return 0; }
> +static inline int ns_cgroup_verify(struct cgroup *cgroup) { return 0; }
> +static inline void copy_hijack_nsproxy(struct task_struct *tsk,
> + struct cgroup *cgroup) {}
> #endif
>
> #endif
> Index: upstream/include/linux/ptrace.h
> ===
> --- upstream.orig/include/linux/ptrace.h
> +++ upstream/include/linux/ptrace.h
> @@ -97,6 +97,7 @@ extern void __ptrace_link(struct task_st
> extern void __ptrace_unlink(struct task_struct *child);
> extern void ptrace_untrace(struct task_struct *child);
> extern int ptrace_may_attach(struct task_struct *task);
> +extern int ptrace_may_attach_locked(struct task_struct *task);
>
> static inline void ptrace_link(struct task_struct *child,
>   struct task_struct *new_parent)
> Index: upstream/include/linux/sched.h
> ===
> --- upstream.orig/include/linux/sched.h
> +++ upstream/include/linux/sched.h
> @@ -29,6 +29,13 @@
> #define CLONE_NEWNET 0x40000000 /* New network namespace */
>
> /*
> + * Hijack flags
> + */
> +#define HIJACK_PID 1 /* 'id' is a pid */
> +#define HIJACK_CGROUP 2 /* 'id' is an open fd for a cgroup dir */
> +#define HIJACK_NS 3 /* 'id' is an open fd for a cgroup dir */

```
> +
> +/*
>   * Scheduling policies
>   */
>  #define SCHED_NORMAL  0
> @@ -1693,9 +1700,19 @@ extern int allow_signal(int);
>  extern int disallow_signal(int);
>
>  extern int do_execve(char *, char __user * __user *, char __user * __user *, struct pt_regs *);
> -extern long do_fork(unsigned long, unsigned long, struct pt_regs *, unsigned long, int __user *,
int __user *);
> +extern long do_fork(unsigned long, unsigned long, struct pt_regs *,
> +  unsigned long, int __user *, int __user *);
>  struct task_struct *fork_idle(int);
>
> +extern int hijack_task(struct task_struct *task, unsigned long clone_flags,
> +  struct pt_regs regs, unsigned long sp);
> +extern int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
> +  unsigned long sp);
> +extern int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
> +  struct pt_regs regs, unsigned long sp);
> +extern int hijack_ns(unsigned int fd, unsigned long clone_flags,
> +  struct pt_regs regs, unsigned long sp);
> +
>  extern void set_task_comm(struct task_struct *tsk, char *from);
>  extern void get_task_comm(char *to, struct task_struct *tsk);
>
> Index: upstream/include/linux/syscalls.h
> ===================================================================
> --- upstream.orig/include/linux/syscalls.h
> +++ upstream/include/linux/syscalls.h
> @@ -614,4 +614,6 @@ asmlinkage long sys_fallocate(int fd, in
>
>  int kernel_execve(const char *filename, char *const argv[], char *const envp[]);
>
> +asmlinkage long sys_hijack(unsigned long flags, int which, unsigned long id);
> +
>  #endif
> Index: upstream/kernel/cgroup.c
> ===================================================================
> --- upstream.orig/kernel/cgroup.c
> +++ upstream/kernel/cgroup.c
> @@ -44,6 +44,7 @@
>  #include <linux/kmod.h>
>  #include <linux/delayacct.h>
>  #include <linux/cgroupstats.h>
> +#include <linux/file.h>
>
```

> #include <asm/atomic.h>
>
> @@ -2442,15 +2443,25 @@ static struct file_operations proc_cgrou
>   * At the point that cgroup_fork() is called, 'current' is the parent
>   * task, and the passed argument 'child' points to the child task.
>   */
> -void cgroup_fork(struct task_struct *child)
> +void cgroup_fork(struct task_struct *parent, struct task_struct *child)
> {
> - task_lock(current);
> - child->cgroups = current->cgroups;
> + if (parent == current)
> + task_lock(parent);
> + child->cgroups = parent->cgroups;
>   get_css_set(child->cgroups);
> - task_unlock(current);
> + if (parent == current)
> + task_unlock(parent);
>   INIT_LIST_HEAD(&child->cg_list);
> }
>
> +void cgroup_fork_fromcgroup(struct cgroup *new_cg, struct task_struct *child)
> +{
> + mutex_lock(&cgroup_mutex);
> + child->cgroups = find_css_set(child->cgroups, new_cg);
> + INIT_LIST_HEAD(&child->cg_list);
> + mutex_unlock(&cgroup_mutex);
> +}
> +
> /**
>   * cgroup_fork_callbacks - called on a new task very soon before
>   * adding it to the tasklist. No need to take any locks since no-one
> @@ -2801,3 +2812,117 @@ static void cgroup_release_agent(struct
>   spin_unlock(&release_list_lock);
>   mutex_unlock(&cgroup_mutex);
> }
> +
> +static inline int task_available(struct task_struct *task)
> +{
> + if (task == current)
> + return 0;
> + if (task_session(task) == task_session(current))
> + return 0;
> + switch (task->state) {
> + case TASK_RUNNING:
> + case TASK_INTERRUPTIBLE:
> + return 1;
> + default:

```
> +   return 0;
> + }
> +}
> +
> +struct cgroup *cgroup_from_fd(unsigned int fd)
> +{
> + struct file *file;
> + struct cgroup *cgroup = NULL;;
> +
> + file = fget(fd);
> + if (!file)
> +  return NULL;
> +
> + if (!file->f_dentry || !file->f_dentry->d_sb)
> +  goto out_fput;
> + if (file->f_dentry->d_parent->d_sb->s_magic != CGROUP_SUPER_MAGIC)
> +  goto out_fput;
> + if (strcmp(file->f_dentry->d_name.name, "tasks"))
> +  goto out_fput;
> +
> + cgroup = __d_cgrp(file->f_dentry->d_parent);
> +
> +out_fput:
> + fput(file);
> + return cgroup;
> +}
> +
> +/*
> + * Takes an integer which is a open fd in current for a valid
> + * cgroupfs file.  Returns a task in that cgroup, with its
> + * refcount bumped.
> + * Since we have an open file on the cgroup tasks file, we
> + * at least don't have to worry about the cgroup being freed
> + * in the middle of this.
> + */
> +struct task_struct *task_from_cgroup_fd(unsigned int fd)
> +{
> + struct cgroup *cgroup;
> + struct cgroup_iter it;
> + struct task_struct *task = NULL;
> +
> + cgroup = cgroup_from_fd(fd);
> + if (!cgroup)
> +  return NULL;
> +
> + rcu_read_lock();
> + cgroup_iter_start(cgroup, &it);
> + do {
```

```
> +  task = cgroup_iter_next(cgroup, &it);
> +  if (task)
> +   printk(KERN_NOTICE "task %d state %lx\n",
> +    task->pid, task->state);
> + } while (task && !task_available(task));
> + cgroup_iter_end(cgroup, &it);
> + if (task)
> +  get_task_struct(task);
> + rcu_read_unlock();
> + return task;
> +}
> +
> +/*
> + * is current allowed to hijack tsk?
> + * permission will also be denied elsewhere if
> + *  current may not ptrace tsk
> + *  security_task_alloc(new_task, tsk) returns -EPERM
> + * Here we are only checking whether current may attach
> + * to tsk's cgroup.  If you can't enter the cgroup, you can't
> + * hijack it.
> + *
> + * XXX TODO This means that ns_cgroup.c will need to allow
> + * entering all descendent cgroups, not just the immediate
> + * child.
> + */
> +int cgroup_may_hijack(struct task_struct *tsk)
> +{
> + int ret = 0;
> + struct cgroupfs_root *root;
> +
> + mutex_lock(&cgroup_mutex);
> + for_each_root(root) {
> +  struct cgroup_subsys *ss;
> +  struct cgroup *cgroup;
> +  int subsys_id;
> +
> +  /* Skip this hierarchy if it has no active subsystems */
> +  if (!root->actual_subsys_bits)
> +   continue;
> +  get_first_subsys(&root->top_cgroup, NULL, &subsys_id);
> +  cgroup = task_cgroup(tsk, subsys_id);
> +  for_each_subsys(root, ss) {
> +   if (ss->may_hijack) {
> +    ret = ss->may_hijack(ss, cgroup, tsk);
> +    if (ret)
> +     goto out_unlock;
> +   }
> +  }
```

```
> + }
> +
> +out_unlock:
> + mutex_unlock(&cgroup_mutex);
> + return ret;
> +}
> Index: upstream/kernel/fork.c
> ===================================================================
> --- upstream.orig/kernel/fork.c
> +++ upstream/kernel/fork.c
> @@ -189,7 +189,7 @@ static struct task_struct *dup_task_stru
>    return NULL;
>   }
>
> - setup_thread_stack(tsk, orig);
> + setup_thread_stack(tsk, current);
>
>  #ifdef CONFIG_CC_STACKPROTECTOR
>   tsk->stack_canary = get_random_int();
> @@ -616,13 +616,14 @@ struct fs_struct *copy_fs_struct(struct
>
>  EXPORT_SYMBOL_GPL(copy_fs_struct);
>
> -static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
> +static inline int copy_fs(unsigned long clone_flags,
> +  struct task_struct *src, struct task_struct *tsk)
>  {
>   if (clone_flags & CLONE_FS) {
> -  atomic_inc(&current->fs->count);
> +  atomic_inc(&src->fs->count);
>    return 0;
>   }
> - tsk->fs = __copy_fs_struct(current->fs);
> + tsk->fs = __copy_fs_struct(src->fs);
>   if (!tsk->fs)
>    return -ENOMEM;
>   return 0;
> @@ -962,6 +963,42 @@ static void rt_mutex_init_task(struct ta
>  #endif
>  }
>
> +void copy_hijackable_taskinfo(struct task_struct *p,
> +  struct task_struct *task)
> +{
> + p->uid = task->uid;
> + p->euid = task->euid;
> + p->suid = task->suid;
> + p->fsuid = task->fsuid;
```

> + p->gid = task->gid;
> + p->egid = task->egid;
> + p->sgid = task->sgid;
> + p->fsgid = task->fsgid;
> + p->cap_effective = task->cap_effective;
> + p->cap_inheritable = task->cap_inheritable;
> + p->cap_permitted = task->cap_permitted;
> + p->keep_capabilities = task->keep_capabilities;
> + p->user = task->user;
> + /*
> + * should keys come from parent or hijack-src?
> + */
> +#ifdef CONFIG_SYSVIPC
> + p->sysvsem = task->sysvsem;
> +#endif
> + p->fs = task->fs;
> + p->nsproxy = task->nsproxy;
> +}
> +
> +#define HIJACK_SOURCE_TASK 1
> +#define HIJACK_SOURCE_CG 2
> +struct hijack_source_info {
> + char type;
> + union hijack_source_union {
> + struct task_struct *task;
> + struct cgroup *cgroup;
> + } u;
> +};
> +
> /*
>   * This creates a new process as a copy of the old one,
>   * but does not actually start it yet.
> @@ -970,7 +1007,8 @@ static void rt_mutex_init_task(struct ta
>   * parts of the process environment (as per the clone
>   * flags). The actual kick-off is left to the caller.
>   */
> -static struct task_struct *copy_process(unsigned long clone_flags,
> +static struct task_struct *copy_process(struct hijack_source_info *src,
> +   unsigned long clone_flags,
>   unsigned long stack_start,
>   struct pt_regs *regs,
>   unsigned long stack_size,
> @@ -980,6 +1018,12 @@ static struct task_struct *copy_process(
>   int retval;
>   struct task_struct *p;
>   int cgroup_callbacks_done = 0;
> + struct task_struct *task;
> +

```
> + if (src->type == HIJACK_SOURCE_TASK)
> +  task = src->u.task;
> + else
> +  task = current;
>
>   if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
>    return ERR_PTR(-EINVAL);
> @@ -1007,6 +1051,10 @@ static struct task_struct *copy_process(
>   p = dup_task_struct(current);
>   if (!p)
>    goto fork_out;
> + if (current != task)
> +  copy_hijackable_taskinfo(p, task);
> + else if (src->type == HIJACK_SOURCE_CG)
> +  copy_hijack_nsproxy(p, src->u.cgroup);
>
>   rt_mutex_init_task(p);
>
> @@ -1084,7 +1132,10 @@ static struct task_struct *copy_process(
>  #endif
>   p->io_context = NULL;
>   p->audit_context = NULL;
> - cgroup_fork(p);
> + if (src->type == HIJACK_SOURCE_CG)
> +  cgroup_fork_fromcgroup(src->u.cgroup, p);
> + else
> +  cgroup_fork(task, p);
>  #ifdef CONFIG_NUMA
>    p->mempolicy = mpol_copy(p->mempolicy);
>    if (IS_ERR(p->mempolicy)) {
> @@ -1135,7 +1186,7 @@ static struct task_struct *copy_process(
>    goto bad_fork_cleanup_audit;
>   if ((retval = copy_files(clone_flags, p)))
>    goto bad_fork_cleanup_semundo;
> - if ((retval = copy_fs(clone_flags, p)))
> + if ((retval = copy_fs(clone_flags, task, p)))
>    goto bad_fork_cleanup_files;
>   if ((retval = copy_sighand(clone_flags, p)))
>    goto bad_fork_cleanup_fs;
> @@ -1167,7 +1218,7 @@ static struct task_struct *copy_process(
>   p->pid = pid_nr(pid);
>   p->tgid = p->pid;
>   if (clone_flags & CLONE_THREAD)
> -  p->tgid = current->tgid;
> +  p->tgid = task->tgid;
>
>   p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr : NULL;
>   /*
```

```
> @@ -1378,8 +1429,12 @@ struct task_struct * __cpuinit fork_idle
>  {
>   struct task_struct *task;
>   struct pt_regs regs;
> + struct hijack_source_info src;
>
> - task = copy_process(CLONE_VM, 0, idle_regs(&regs), 0, NULL,
> + src.type = HIJACK_SOURCE_TASK;
> + src.u.task = current;
> +
> + task = copy_process(&src, CLONE_VM, 0, idle_regs(&regs), 0, NULL,
>       &init_struct_pid);
>   if (!IS_ERR(task))
>    init_idle(task, cpu);
> @@ -1404,29 +1459,43 @@ static int fork_traceflag(unsigned clone
>  }
>
>  /*
> - *  Ok, this is the main fork-routine.
> - *
> - * It copies the process, and if successful kick-starts
> - * it and waits for it to finish using the VM if required.
> + * if called with task!=current, then caller must ensure that
> + *    1. it has a reference to task
> + *    2. current must have ptrace permission to task
>   */
> -long do_fork(unsigned long clone_flags,
> +long do_fork_task(struct hijack_source_info *src,
> +  unsigned long clone_flags,
>         unsigned long stack_start,
>         struct pt_regs *regs,
>         unsigned long stack_size,
>         int __user *parent_tidptr,
>         int __user *child_tidptr)
>  {
> - struct task_struct *p;
> + struct task_struct *p, *task;
>   int trace = 0;
>   long nr;
>
> + if (src->type == HIJACK_SOURCE_TASK)
> +  task = src->u.task;
> + else
> +  task = current;
> + if (task != current) {
> +  /* sanity checks */
> +  /* we only want to allow hijacking the simplest cases */
> +  if (clone_flags & CLONE_SYSVSEM)
```

```
> +   return -EINVAL;
> +  if (current->ptrace)
> +   return -EPERM;
> +  if (task->ptrace)
> +   return -EINVAL;
> + }
>   if (unlikely(current->ptrace)) {
>    trace = fork_traceflag (clone_flags);
>    if (trace)
>     clone_flags |= CLONE_PTRACE;
>   }
>
> - p = copy_process(clone_flags, stack_start, regs, stack_size,
> + p = copy_process(src, clone_flags, stack_start, regs, stack_size,
>     child_tidptr, NULL);
>   /*
>    * Do this prior waking up the new thread - the thread pointer
> @@ -1484,6 +1553,106 @@ long do_fork(unsigned long clone_flags,
>   return nr;
>  }
>
> +/*
> + *  Ok, this is the main fork-routine.
> + *
> + * It copies the process, and if successful kick-starts
> + * it and waits for it to finish using the VM if required.
> + */
> +long do_fork(unsigned long clone_flags,
> +       unsigned long stack_start,
> +       struct pt_regs *regs,
> +       unsigned long stack_size,
> +       int __user *parent_tidptr,
> +       int __user *child_tidptr)
> +{
> + struct hijack_source_info src = {
> +   .type = HIJACK_SOURCE_TASK,
> +   .u = { .task = current, },
> + };
> + return do_fork_task(&src, clone_flags, stack_start,
> +  regs, stack_size, parent_tidptr, child_tidptr);
> +}
> +
> +/*
> + * Called with task count bumped, drops task count before returning
> + */
> +int hijack_task(struct task_struct *task, unsigned long clone_flags,
> +  struct pt_regs regs, unsigned long sp)
> +{
```

```
> + int ret = -EPERM;
> + struct hijack_source_info src = {
> +    .type = HIJACK_SOURCE_TASK,
> +    .u = { .task = task, },
> + };
> +
> + task_lock(task);
> + put_task_struct(task);
> + if (!ptrace_may_attach_locked(task))
> +  goto out_unlock_task;
> + if (task == current)
> +  goto out_unlock_task;
> + ret = cgroup_may_hijack(task);
> + if (ret)
> +  goto out_unlock_task;
> + if (task->ptrace) {
> +  ret = -EBUSY;
> +  goto out_unlock_task;
> + }
> + ret = do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
> +
> +out_unlock_task:
> + task_unlock(task);
> + return ret;
> +}
> +
> +int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
> +        unsigned long sp)
> +{
> + struct task_struct *task;
> +
> + rcu_read_lock();
> + task = find_task_by_vpid(pid);
> + if (task)
> +  get_task_struct(task);
> + rcu_read_unlock();
> +
> + if (!task)
> +  return -EINVAL;
> +
> + return hijack_task(task, clone_flags, regs, sp);
> +}
> +
> +int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
> +     struct pt_regs regs, unsigned long sp)
> +{
> + struct task_struct *task;
> +
```

> + task = task_from_cgroup_fd(fd);
> + if (!task)
> + return -EINVAL;
> +
> + return hijack_task(task, clone_flags, regs, sp);
> +}
> +
> +int hijack_ns(unsigned int fd, unsigned long clone_flags,
> +   struct pt_regs regs, unsigned long sp)
> +{
> + struct hijack_source_info src;
> + struct cgroup *cgroup;
> +
> + cgroup = cgroup_from_fd(fd);
> + if (!cgroup)
> + return -EINVAL;
> +
> + if (!ns_cgroup_verify(cgroup))
> + return -EINVAL;
> +
> + src.type = HIJACK_SOURCE_CG;
> + src.u.cgroup = cgroup;
> + return do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
> +}
> +
> #ifndef ARCH_MIN_MMSTRUCT_ALIGN
> #define ARCH_MIN_MMSTRUCT_ALIGN 0
> #endif
> Index: upstream/kernel/ns_cgroup.c
> ===
> --- upstream.orig/kernel/ns_cgroup.c
> +++ upstream/kernel/ns_cgroup.c
> @@ -7,9 +7,11 @@
> #include <linux/module.h>
> #include <linux/cgroup.h>
> #include <linux/fs.h>
> +#include <linux/nsproxy.h>
>
> struct ns_cgroup {
>   struct cgroup_subsys_state css;
> + struct nsproxy *nsproxy;
>   spinlock_t lock;
> };
>
> @@ -22,9 +24,51 @@ static inline struct ns_cgroup *cgroup_t
>   struct ns_cgroup, css);
> }
>

```
> -int ns_cgroup_clone(struct task_struct *task)
> +int ns_cgroup_clone(struct task_struct *task, struct nsproxy *nsproxy)
>  {
> - return cgroup_clone(task, &ns_subsys);
> + struct cgroup *cgroup;
> + struct ns_cgroup *ns_cgroup;
> + int ret = cgroup_clone(task, &ns_subsys);
> +
> + if (ret)
> +  return ret;
> +
> + cgroup = task_cgroup(task, ns_subsys_id);
> + ns_cgroup = cgroup_to_ns(cgroup);
> + ns_cgroup->nsproxy = nsproxy;
> + get_nsproxy(nsproxy);
> +
> + return 0;
> +}
> +
> +int ns_cgroup_verify(struct cgroup *cgroup)
> +{
> + struct cgroup_subsys_state *css;
> + struct ns_cgroup *ns_cgroup;
> +
> + css = cgroup_subsys_state(cgroup, ns_subsys_id);
> + if (!css)
> +  return 0;
> + ns_cgroup = container_of(css, struct ns_cgroup, css);
> + if (!ns_cgroup->nsproxy)
> +  return 0;
> + return 1;
> +}
> +
> +/*
> + * this shouldn't be called unless ns_cgroup_verify() has
> + * confirmed that there is a ns_cgroup in this cgroup
> + *
> + * tsk is not yet running, and has not yet taken a reference
> + * to it's previous ->nsproxy, so we just do a simple assignment
> + * rather than switch_task_namespaces()
> + */
> +void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup)
> +{
> + struct ns_cgroup *ns_cgroup;
> +
> + ns_cgroup = cgroup_to_ns(cgroup);
> + tsk->nsproxy = ns_cgroup->nsproxy;
>  }
```

>
> /*
> @@ -60,6 +104,42 @@ static int ns_can_attach(struct cgroup_s
>   return 0;
> }
>
> +static void ns_attach(struct cgroup_subsys *ss,
> +   struct cgroup *cgroup, struct cgroup *oldcgroup,
> +   struct task_struct *tsk)
> +{
> + struct ns_cgroup *ns_cgroup = cgroup_to_ns(cgroup);
> +
> + if (likely(ns_cgroup->nsproxy))
> + return;
> +
> + spin_lock(&ns_cgroup->lock);
> + if (!ns_cgroup->nsproxy) {
> + ns_cgroup->nsproxy = tsk->nsproxy;
> + get_nsproxy(ns_cgroup->nsproxy);
> + }
> + spin_unlock(&ns_cgroup->lock);
> +}
> +
> +/*
> + * only allow hijacking child namespaces
> + * Q: is it crucial to prevent hijacking a task in your same cgroup?
> + */
> +static int ns_may_hijack(struct cgroup_subsys *ss,
> + struct cgroup *new_cgroup, struct task_struct *task)
> +{
> + if (current == task)
> + return -EINVAL;
> +
> + if (!capable(CAP_SYS_ADMIN))
> + return -EPERM;
> +
> + if (!cgroup_is_descendant(new_cgroup))
> + return -EPERM;
> +
> + return 0;
> +}
> +
> /*
>   * Rules: you can only create a cgroup if
>   *   1. you are capable(CAP_SYS_ADMIN)
> @@ -88,12 +168,16 @@ static void ns_destroy(struct cgroup_sub
>   struct ns_cgroup *ns_cgroup;
>

```
>   ns_cgroup = cgroup_to_ns(cgroup);
> + if (ns_cgroup->nsproxy)
> +  put_nsproxy(ns_cgroup->nsproxy);
>   kfree(ns_cgroup);
>  }
>
>  struct cgroup_subsys ns_subsys = {
>   .name = "ns",
>   .can_attach = ns_can_attach,
> + .attach = ns_attach,
> + .may_hijack = ns_may_hijack,
>   .create = ns_create,
>   .destroy  = ns_destroy,
>   .subsys_id = ns_subsys_id,
> Index: upstream/kernel/nsproxy.c
> ===================================================================
> --- upstream.orig/kernel/nsproxy.c
> +++ upstream/kernel/nsproxy.c
> @@ -144,7 +144,7 @@ int copy_namespaces(unsigned long flags,
>    goto out;
>   }
>
> - err = ns_cgroup_clone(tsk);
> + err = ns_cgroup_clone(tsk, new_ns);
>   if (err) {
>    put_nsproxy(new_ns);
>    goto out;
> @@ -196,7 +196,7 @@ int unshare_nsproxy_namespaces(unsigned
>    goto out;
>   }
>
> - err = ns_cgroup_clone(current);
> + err = ns_cgroup_clone(current, *new_nsp);
>   if (err)
>    put_nsproxy(*new_nsp);
>
> Index: upstream/kernel/ptrace.c
> ===================================================================
> --- upstream.orig/kernel/ptrace.c
> +++ upstream/kernel/ptrace.c
> @@ -159,6 +159,13 @@ int ptrace_may_attach(struct task_struct
>   return !err;
>  }
>
> +int ptrace_may_attach_locked(struct task_struct *task)
> +{
> + int err;
> + err = may_attach(task);
```

> + return !err;
> +}
> +
> int ptrace_attach(struct task_struct *task)
> {
>   int retval;
> -
> To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
>

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
  Itanium. Vista. GPLv3. Complexity at work

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by rodrigo on Tue, 27 Nov 2007 11:08:31 GMT
View Forum Message <> Reply to Message

It will give another easy way to locate selinux security structures inside
the kernel, will not?

Again, if you have a kernel vulnerability and this feature, someone will
easily disable selinux for the process, or just change the security concerns
for it ;).

cya,

Rodrigo (BSDaemon).

--
http://www.kernelhacking.com/rodrigo

Kernel Hacking: If i really know, i can hack

GPG KeyID: 1FCEDEA1

--------- Mensagem Original --------
De: Joshua Brindle <method@manicmethod.com>
Para: Mark Nelson <markn@au1.ibm.com>

linux-security-module@vger.kernel.org, selinux@tycho.nsa.gov,
menage@google.com, Stephen Smalley <sds@tycho.nsa.gov>, James Morris
<jmorris@namei.org>, Serge E. Hallyn <serue@us.ibm.com>
Assunto: Re: [PATCH 2/2] hijack: update task_alloc_security
Data: 27/11/07 02:38

>
> Mark Nelson wrote:
> > Subject: [PATCH 2/2] hijack: update task_alloc_security
> >
> > Update task_alloc_security() to take the hijacked task as a second
> > argument.
> >
> > For the selinux version, refuse permission if hijack_src!=current,
> > since we have no idea what the proper behavior is. Even if we
> > assume that the resulting child should be in the hijacked task's
> > domain, depending on the policy that may not be enough information
> > since init_t executing /bin/bash could result in a different domain
> > than login_t executing /bin/bash.
> >
> >
> This means its basically not possible to hijack tasks with SELinux
> right? It would be a shame if this weren't useful to people running
SELinux.
>
> It seems to me (I may be wrong, I'm sure someone will let me know if I
> am) that the right way to handle this with SELinux is to check to see if
> the current task (caller of sys_hijack) has permission to ptrace (or
> some other permission deemed suitable, perhaps a new one) and if so copy
> the security blob pointer from the hijacked task to the new one (we
> don't want tranquility problems).
>
> From your paragraph above it seems like you were thinking there should
> be a transition at hijack time but we don't automatically transition
> anywhere except exec.
>
> Anyway, I just don't think you should completely disable this for
> SELinux users.
>
> > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > ---
> > include/linux/security.h | 12 ++++++++++---

> > kernel/fork.c | 2 +-
> > security/dummy.c | 3 ++-
> > security/security.c | 4 ++--
> > security/selinux/hooks.c | 6 +++++-
> > 5 files changed, 19 insertions(+), 8 deletions(-)
> >
> > Index: upstream/include/linux/security.h
> > ===
> > --- upstream.orig/include/linux/security.h
> > +++ upstream/include/linux/security.h
> > @@ -545,9 +545,13 @@ struct request_sock;
> >   * Return 0 if permission is granted.
> >   * @task_alloc_security:
> >   * @p contains the task_struct for child process.
> > + * @task contains the task_struct for process to be hijacked
> >   * Allocate and attach a security structure to the p->security
field. The
> >   * security field is initialized to NULL when the task structure is
> >   * allocated.
> > + * @task will usually be current. If it is not equal to current,
then
> > + * a sys_hijack system call is going on, and current is asking for a
> > + * child to be created in the context of the hijack src, @task.
> >   * Return 0 if operation was successful.
> >   * @task_free_security:
> >   * @p contains the task_struct for process.
> > @@ -1301,7 +1305,8 @@ struct security_operations {
> >   int (*dentry_open) (struct file *file);
> >
> >   int (*task_create) (unsigned long clone_flags);
> > - int (*task_alloc_security) (struct task_struct * p);
> > + int (*task_alloc_security) (struct task_struct *p,
> > +   struct task_struct *task);
> >   void (*task_free_security) (struct task_struct * p);
> >   int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> >   int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> > @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> > int security_file_receive(struct file *file);
> > int security_dentry_open(struct file *file);
> > int security_task_create(unsigned long clone_flags);
> > -int security_task_alloc(struct task_struct *p);
> > +int security_task_alloc(struct task_struct *p, struct task_struct
*task);
> > void security_task_free(struct task_struct *p);
> > int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int
flags);
> > int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> > @@ -2021,7 +2026,8 @@ static inline int security_task_create (

> > return 0;
> > }
> >
> > -static inline int security_task_alloc (struct task_struct *p)
> > +static inline int security_task_alloc(struct task_struct *p,
> > + struct task_struct *task)
> > {
> > return 0;
> > }
> > Index: upstream/kernel/fork.c
> > ===
> > --- upstream.orig/kernel/fork.c
> > +++ upstream/kernel/fork.c
> > @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
> > /* Perform scheduler related setup. Assign this task to a CPU. */
> > sched_fork(p, clone_flags);
> >
> > - if ((retval = security_task_alloc(p)))
> > + if ((retval = security_task_alloc(p, task)))
> > goto bad_fork_cleanup_policy;
> > if ((retval = audit_alloc(p)))
> > goto bad_fork_cleanup_security;
> > Index: upstream/security/dummy.c
> > ===
> > --- upstream.orig/security/dummy.c
> > +++ upstream/security/dummy.c
> > @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
> > return 0;
> > }
> >
> > -static int dummy_task_alloc_security (struct task_struct *p)
> > +static int dummy_task_alloc_security(struct task_struct *p,
> > + struct task_struct *task)
> > {
> > return 0;
> > }
> > Index: upstream/security/security.c
> > ===
> > --- upstream.orig/security/security.c
> > +++ upstream/security/security.c
> > @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
> > return security_ops->task_create(clone_flags);
> > }
> >
> > -int security_task_alloc(struct task_struct *p)
> > +int security_task_alloc(struct task_struct *p, struct task_struct
*task)
> > {

> > - return security_ops->task_alloc_security(p);
> > + return security_ops->task_alloc_security(p, task);
> > }
> >
> > void security_task_free(struct task_struct *p)
> > Index: upstream/security/selinux/hooks.c
> > ===
> > --- upstream.orig/security/selinux/hooks.c
> > +++ upstream/security/selinux/hooks.c
> > @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
> >   return task_has_perm(current, current, PROCESS__FORK);
> > }
> >
> > -static int selinux_task_alloc_security(struct task_struct *tsk)
> > +static int selinux_task_alloc_security(struct task_struct *tsk,
> > +   struct task_struct *hijack_src)
> > {
> >   struct task_security_struct *tsec1, *tsec2;
> >   int rc;
> >
> > + if (hijack_src != current)
> > + return -EPERM;
> > +
> >   tsec1 = current->security;
> >
> >   rc = task_alloc_security(tsk);
> > -
> > To unsubscribe from this list: send the line "unsubscribe
linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
>
>
> -
> To unsubscribe from this list: send the line "unsubscribe
linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
>
>
>
>

_____
Message sent using UebiMiau 2.7.2

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Stephen Smalley on Tue, 27 Nov 2007 14:36:28 GMT
View Forum Message <> Reply to Message

On Tue, 2007-11-27 at 00:52 -0500, Joshua Brindle wrote:
> Mark Nelson wrote:
> > Subject: [PATCH 2/2] hijack: update task_alloc_security
> >
> > Update task_alloc_security() to take the hijacked task as a second
> > argument.
> >
> > For the selinux version, refuse permission if hijack_src!=current,
> > since we have no idea what the proper behavior is. Even if we
> > assume that the resulting child should be in the hijacked task's
> > domain, depending on the policy that may not be enough information
> > since init_t executing /bin/bash could result in a different domain
> > than login_t executing /bin/bash.
> >
> >
> This means its basically not possible to hijack tasks with SELinux
> right? It would be a shame if this weren't useful to people running SELinux.

I agree with this part - we don't want people to have to choose between
using containers and using selinux, so if hijack is going to be a
requirement for effective use of containers, then we need to make them
work together.

> It seems to me (I may be wrong, I'm sure someone will let me know if I
> am) that the right way to handle this with SELinux is to check to see if
> the current task (caller of sys_hijack) has permission to ptrace (or

I think this may already happen in the first patch, by virtue of calling
the existing ptrace checks including the security hook. Right?

> some other permission deemed suitable, perhaps a new one) and if so copy
> the security blob pointer from the hijacked task to the new one (we
> don't want tranquility problems).

Just to clarify, we wouldn't be copying the pointer; here we are
allocating and populating a new task's security structure. We can
either continue to inherit the SIDs from current in all cases, or we

could set tsec1 = hijack_src->security; in selinux_task_alloc_security()
if we wanted to inherit from the hijacked task instead. The latter
would be similar to what you do in copy_hijackable_taskinfo() for uids
and capabilities IIUC. However, which behavior is right needs more
discussion I think, as the new task is a mixture of the caller's state
and the hijacked task's state. Which largely seems a recipe for
disaster.

> From your paragraph above it seems like you were thinking there should
> be a transition at hijack time but we don't automatically transition
> anywhere except exec.
>
> Anyway, I just don't think you should completely disable this for
> SELinux users.
>
> > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > ---
> > include/linux/security.h | 12 ++++++++---
> > kernel/fork.c | 2 +-
> > security/dummy.c | 3 ++-
> > security/security.c | 4 ++--
> > security/selinux/hooks.c | 6 +++++-
> > 5 files changed, 19 insertions(+), 8 deletions(-)
> >
> > Index: upstream/include/linux/security.h
> > ===
> > --- upstream.orig/include/linux/security.h
> > +++ upstream/include/linux/security.h
> > @@ -545,9 +545,13 @@ struct request_sock;
> > * Return 0 if permission is granted.
> > * @task_alloc_security:
> > * @p contains the task_struct for child process.
> > + * @task contains the task_struct for process to be hijacked
> > * Allocate and attach a security structure to the p->security field. The
> > * security field is initialized to NULL when the task structure is
> > * allocated.
> > + * @task will usually be current. If it is not equal to current, then
> > + * a sys_hijack system call is going on, and current is asking for a
> > + * child to be created in the context of the hijack src, @task.
> > * Return 0 if operation was successful.
> > * @task_free_security:
> > * @p contains the task_struct for process.
> > @@ -1301,7 +1305,8 @@ struct security_operations {
> > int (*dentry_open) (struct file *file);
> >
> > int (*task_create) (unsigned long clone_flags);
> > - int (*task_alloc_security) (struct task_struct * p);

```
> > + int (*task_alloc_security) (struct task_struct *p,
> > +        struct task_struct *task);
> >   void (*task_free_security) (struct task_struct * p);
> >   int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> >   int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> > @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> >  int security_file_receive(struct file *file);
> >  int security_dentry_open(struct file *file);
> >  int security_task_create(unsigned long clone_flags);
> > -int security_task_alloc(struct task_struct *p);
> > +int security_task_alloc(struct task_struct *p, struct task_struct *task);
> >  void security_task_free(struct task_struct *p);
> >  int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int flags);
> >  int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> > @@ -2021,7 +2026,8 @@ static inline int security_task_create (
> >   return 0;
> >  }
> >
> > -static inline int security_task_alloc (struct task_struct *p)
> > +static inline int security_task_alloc(struct task_struct *p,
> > +           struct task_struct *task)
> >  {
> >   return 0;
> >  }
> > Index: upstream/kernel/fork.c
> > ===================================================================
> > --- upstream.orig/kernel/fork.c
> > +++ upstream/kernel/fork.c
> > @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
> >   /* Perform scheduler related setup. Assign this task to a CPU. */
> >   sched_fork(p, clone_flags);
> >
> > - if ((retval = security_task_alloc(p)))
> > + if ((retval = security_task_alloc(p, task)))
> >    goto bad_fork_cleanup_policy;
> >   if ((retval = audit_alloc(p)))
> >    goto bad_fork_cleanup_security;
> > Index: upstream/security/dummy.c
> > ===================================================================
> > --- upstream.orig/security/dummy.c
> > +++ upstream/security/dummy.c
> > @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
> >   return 0;
> >  }
> >
> > -static int dummy_task_alloc_security (struct task_struct *p)
> > +static int dummy_task_alloc_security(struct task_struct *p,
> > +           struct task_struct *task)
```

> > {
> > return 0;
> > }
> > Index: upstream/security/security.c
> > ===
> > --- upstream.orig/security/security.c
> > +++ upstream/security/security.c
> > @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
> > return security_ops->task_create(clone_flags);
> > }
> >
> > -int security_task_alloc(struct task_struct *p)
> > +int security_task_alloc(struct task_struct *p, struct task_struct *task)
> > {
> > - return security_ops->task_alloc_security(p);
> > + return security_ops->task_alloc_security(p, task);
> > }
> >
> > void security_task_free(struct task_struct *p)
> > Index: upstream/security/selinux/hooks.c
> > ===
> > --- upstream.orig/security/selinux/hooks.c
> > +++ upstream/security/selinux/hooks.c
> > @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
> > return task_has_perm(current, current, PROCESS__FORK);
> > }
> >
> > -static int selinux_task_alloc_security(struct task_struct *tsk)
> > +static int selinux_task_alloc_security(struct task_struct *tsk,
> > + struct task_struct *hijack_src)
> > {
> > struct task_security_struct *tsec1, *tsec2;
> > int rc;
> >
> > + if (hijack_src != current)
> > + return -EPERM;
> > +
> > tsec1 = current->security;
> >
> > rc = task_alloc_security(tsk);
> > -
> > To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
>
>

> -
> To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
--
Stephen Smalley
National Security Agency

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Tue, 27 Nov 2007 15:43:56 GMT
View Forum Message <> Reply to Message

Quoting Stephen Smalley (sds@tycho.nsa.gov):
> On Tue, 2007-11-27 at 00:52 -0500, Joshua Brindle wrote:
> > Mark Nelson wrote:
> > > Subject: [PATCH 2/2] hijack: update task_alloc_security
> > >
> > > Update task_alloc_security() to take the hijacked task as a second
> > > argument.
> > >
> > > For the selinux version, refuse permission if hijack_src!=current,
> > > since we have no idea what the proper behavior is. Even if we
> > > assume that the resulting child should be in the hijacked task's
> > > domain, depending on the policy that may not be enough information
> > > since init_t executing /bin/bash could result in a different domain
> > > than login_t executing /bin/bash.
> > >
> > >
> > This means its basically not possible to hijack tasks with SELinux
> > right? It would be a shame if this weren't useful to people running SELinux.
>
> I agree with this part - we don't want people to have to choose between
> using containers and using selinux, so if hijack is going to be a
> requirement for effective use of containers, then we need to make them
> work together.

Absolutely, we just need to decide how to properly make it work with
selinux. Maybe we check for

 allow (current_domain):(hijacked_process_domain) hijack
 type_transition hijacked_process_domain \
  vserver_enter_binary_t:process vserver1_hijack_admin_t;

The reason I decided to punt on this altogether is that the decision of which type to use for the resulting process seems tricky and definately requires new policy.

If we are using completely custom and simple policies for vservers, i.e. every process in a vserver is type vserver_3_t, then it's not so complicated. But if I hijack a process in 'container 3', and the resulting type is calculated in part based on the type of the hijacked process, then I'll end up with a different type based on whether I'm hijackign the init process, an ssh daemon, someone's shell, etc. Then I do some admin activity in that resulting process and end up labeling files in the vserver based on the process I had hijacked.

> > It seems to me (I may be wrong, I'm sure someone will let me know if I
> > am) that the right way to handle this with SELinux is to check to see if
> > the current task (caller of sys_hijack) has permission to ptrace (or
>
> I think this may already happen in the first patch, by virtue of calling
> the existing ptrace checks including the security hook. Right?

Good point, I forgot about that. So that may suffice for a permissions check. Now we just need to decide how to determine a type for the resulting process.

> > some other permission deemed suitable, perhaps a new one) and if so copy
> > the security blob pointer from the hijacked task to the new one (we
> > don't want tranquility problems).
>
> Just to clarify, we wouldn't be copying the pointer; here we are
> allocating and populating a new task's security structure. We can
> either continue to inherit the SIDs from current in all cases, or we
> could set tsec1 = hijack_src->security; in selinux_task_alloc_security()
> if we wanted to inherit from the hijacked task instead. The latter
> would be similar to what you do in copy_hijackable_taskinfo() for uids
> and capabilities IIUC. However, which behavior is right needs more
> discussion I think, as the new task is a mixture of the caller's state
> and the hijacked task's state. Which largely seems a recipe for
> disaster.

Yes, especially if we don't have a single type for all processes in a container.

Maybe what's needed is a more general discussion about how selinux policy is to be used with containers, even without hijack. I have a vm set aside where I'm starting to write a container policy module. I just need to get a kernel lined up. Was hoping to start next week.

In the meantime I still think it's prudent to punt on the selinux
hijacking code for now.

thanks,
-serge

>
> > From your paragraph above it seems like you were thinking there should
> > be a transition at hijack time but we don't automatically transition
> > anywhere except exec.
> >
> > Anyway, I just don't think you should completely disable this for
> > SELinux users.
> >
> > > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > > ---
> > > include/linux/security.h | 12 ++++++++---
> > > kernel/fork.c | 2 +-
> > > security/dummy.c | 3 ++-
> > > security/security.c | 4 ++--
> > > security/selinux/hooks.c | 6 +++++-
> > > 5 files changed, 19 insertions(+), 8 deletions(-)
> > >
> > > Index: upstream/include/linux/security.h
> > > ===
> > > --- upstream.orig/include/linux/security.h
> > > +++ upstream/include/linux/security.h
> > > @@ -545,9 +545,13 @@ struct request_sock;
> > >   * Return 0 if permission is granted.
> > >   * @task_alloc_security:
> > >   * @p contains the task_struct for child process.
> > > + * @task contains the task_struct for process to be hijacked
> > >   * Allocate and attach a security structure to the p->security field. The
> > >   * security field is initialized to NULL when the task structure is
> > >   * allocated.
> > > + * @task will usually be current. If it is not equal to current, then
> > > + * a sys_hijack system call is going on, and current is asking for a
> > > + * child to be created in the context of the hijack src, @task.
> > >   * Return 0 if operation was successful.
> > >   * @task_free_security:
> > >   * @p contains the task_struct for process.
> > > @@ -1301,7 +1305,8 @@ struct security_operations {
> > >   int (*dentry_open) (struct file *file);
> > >
> > >   int (*task_create) (unsigned long clone_flags);
> > > - int (*task_alloc_security) (struct task_struct * p);
> > > + int (*task_alloc_security) (struct task_struct *p,

```
> > > +        struct task_struct *task);
> > >    void (*task_free_security) (struct task_struct * p);
> > >    int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> > >    int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> > > @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> > >  int security_file_receive(struct file *file);
> > >  int security_dentry_open(struct file *file);
> > >  int security_task_create(unsigned long clone_flags);
> > > -int security_task_alloc(struct task_struct *p);
> > > +int security_task_alloc(struct task_struct *p, struct task_struct *task);
> > >  void security_task_free(struct task_struct *p);
> > >  int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int flags);
> > >  int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> > > @@ -2021,7 +2026,8 @@ static inline int security_task_create (
> > >    return 0;
> > >  }
> > >
> > > -static inline int security_task_alloc (struct task_struct *p)
> > > +static inline int security_task_alloc(struct task_struct *p,
> > > +           struct task_struct *task)
> > >  {
> > >    return 0;
> > >  }
> > > Index: upstream/kernel/fork.c
> > > ===================================================================
> > > --- upstream.orig/kernel/fork.c
> > > +++ upstream/kernel/fork.c
> > > @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
> > >    /* Perform scheduler related setup. Assign this task to a CPU. */
> > >    sched_fork(p, clone_flags);
> > >
> > > - if ((retval = security_task_alloc(p)))
> > > + if ((retval = security_task_alloc(p, task)))
> > >     goto bad_fork_cleanup_policy;
> > >    if ((retval = audit_alloc(p)))
> > >     goto bad_fork_cleanup_security;
> > > Index: upstream/security/dummy.c
> > > ===================================================================
> > > --- upstream.orig/security/dummy.c
> > > +++ upstream/security/dummy.c
> > > @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
> > >    return 0;
> > >  }
> > >
> > > -static int dummy_task_alloc_security (struct task_struct *p)
> > > +static int dummy_task_alloc_security(struct task_struct *p,
> > > +           struct task_struct *task)
> > >  {
```

```
> > >   return 0;
> > >  }
> > > Index: upstream/security/security.c
> > > ===================================================================
> > > --- upstream.orig/security/security.c
> > > +++ upstream/security/security.c
> > > @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
> > >   return security_ops->task_create(clone_flags);
> > >  }
> > >
> > > -int security_task_alloc(struct task_struct *p)
> > > +int security_task_alloc(struct task_struct *p, struct task_struct *task)
> > >  {
> > > - return security_ops->task_alloc_security(p);
> > > + return security_ops->task_alloc_security(p, task);
> > >  }
> > >
> > >  void security_task_free(struct task_struct *p)
> > > Index: upstream/security/selinux/hooks.c
> > > ===================================================================
> > > --- upstream.orig/security/selinux/hooks.c
> > > +++ upstream/security/selinux/hooks.c
> > > @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
> > >   return task_has_perm(current, current, PROCESS__FORK);
> > >  }
> > >
> > > -static int selinux_task_alloc_security(struct task_struct *tsk)
> > > +static int selinux_task_alloc_security(struct task_struct *tsk,
> > > +          struct task_struct *hijack_src)
> > >  {
> > >   struct task_security_struct *tsec1, *tsec2;
> > >   int rc;
> > >
> > > + if (hijack_src != current)
> > > +  return -EPERM;
> > > +
> > >   tsec1 = current->security;
> > >
> > >   rc = task_alloc_security(tsk);
> > > -
> > > To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> > > the body of a message to majordomo@vger.kernel.org
> > > More majordomo info at  http://vger.kernel.org/majordomo-info.html
> > >
> > >
> >
> >
> > -
```

> > To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> --
> Stephen Smalley
> National Security Agency

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Tue, 27 Nov 2007 15:50:13 GMT
View Forum Message <> Reply to Message

Quoting Rodrigo Rubira Branco (BSDaemon) (rodrigo@kernelhacking.com):
> It will give another easy way to locate selinux security structures inside
> the kernel, will not?

By locate, you mean actually finding the structures in kernel memory to
overwrite them?

> Again, if you have a kernel vulnerability and this feature, someone will
> easily disable selinux for the process, or just change the security concerns
> for it ;).

Maybe hijack should go under a kernel config for security reasons.

thanks,
-serge

> cya,
>
>
> Rodrigo (BSDaemon).
>
> --
> http://www.kernelhacking.com/rodrigo
>
> Kernel Hacking: If i really know, i can hack
>
> GPG KeyID: 1FCEDEA1
>
>
> --------- Mensagem Original --------
> De: Joshua Brindle <method@manicmethod.com>
> Para: Mark Nelson <markn@au1.ibm.com>

> C?pia: containers@lists.linux-foundation.org,
> linux-security-module@vger.kernel.org, selinux@tycho.nsa.gov,
> menage@google.com, Stephen Smalley <sds@tycho.nsa.gov>, James Morris
> <jmorris@namei.org>, Serge E. Hallyn <serue@us.ibm.com>
> Assunto: Re: [PATCH 2/2] hijack: update task_alloc_security
> Data: 27/11/07 02:38
>
> >
> > Mark Nelson wrote:
> > > Subject: [PATCH 2/2] hijack: update task_alloc_security
> > >
> > > Update task_alloc_security() to take the hijacked task as a second
> > > argument.
> > >
> > > For the selinux version, refuse permission if hijack_src!=current,
> > > since we have no idea what the proper behavior is. Even if we
> > > assume that the resulting child should be in the hijacked task's
> > > domain, depending on the policy that may not be enough information
> > > since init_t executing /bin/bash could result in a different domain
> > > than login_t executing /bin/bash.
> > >
> > >
> > This means its basically not possible to hijack tasks with SELinux
> > right? It would be a shame if this weren't useful to people running
> SELinux.
> >
> > It seems to me (I may be wrong, I'm sure someone will let me know if I
> > am) that the right way to handle this with SELinux is to check to see if
> > the current task (caller of sys_hijack) has permission to ptrace (or
> > some other permission deemed suitable, perhaps a new one) and if so copy
> > the security blob pointer from the hijacked task to the new one (we
> > don't want tranquility problems).
> >
> > From your paragraph above it seems like you were thinking there should
> > be a transition at hijack time but we don't automatically transition
> > anywhere except exec.
> >
> > Anyway, I just don't think you should completely disable this for
> > SELinux users.
> >
> > > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > > ---
> > > include/linux/security.h | 12 ++++++++---
> > > kernel/fork.c | 2 +-
> > > security/dummy.c | 3 ++-
> > > security/security.c | 4 ++--
> > > security/selinux/hooks.c | 6 +++++-

> > > 5 files changed, 19 insertions(+), 8 deletions(-)
> > >
> > > Index: upstream/include/linux/security.h
> > > ===
> > > --- upstream.orig/include/linux/security.h
> > > +++ upstream/include/linux/security.h
> > > @@ -545,9 +545,13 @@ struct request_sock;
> > >   * Return 0 if permission is granted.
> > >   * @task_alloc_security:
> > >   * @p contains the task_struct for child process.
> > > + * @task contains the task_struct for process to be hijacked
> > >   * Allocate and attach a security structure to the p->security
> field. The
> > >   * security field is initialized to NULL when the task structure is
> > >   * allocated.
> > > + * @task will usually be current. If it is not equal to current,
> then
> > > + * a sys_hijack system call is going on, and current is asking for a
> > > + * child to be created in the context of the hijack src, @task.
> > >   * Return 0 if operation was successful.
> > >   * @task_free_security:
> > >   * @p contains the task_struct for process.
> > > @@ -1301,7 +1305,8 @@ struct security_operations {
> > >   int (*dentry_open) (struct file *file);
> > >
> > >   int (*task_create) (unsigned long clone_flags);
> > > - int (*task_alloc_security) (struct task_struct * p);
> > > + int (*task_alloc_security) (struct task_struct *p,
> > > +   struct task_struct *task);
> > >   void (*task_free_security) (struct task_struct * p);
> > >   int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> > >   int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> > > @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> > > int security_file_receive(struct file *file);
> > > int security_dentry_open(struct file *file);
> > > int security_task_create(unsigned long clone_flags);
> > > -int security_task_alloc(struct task_struct *p);
> > > +int security_task_alloc(struct task_struct *p, struct task_struct
> *task);
> > > void security_task_free(struct task_struct *p);
> > > int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int
> flags);
> > > int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> > > @@ -2021,7 +2026,8 @@ static inline int security_task_create (
> > >   return 0;
> > > }
> > >
> > > -static inline int security_task_alloc (struct task_struct *p)

> > > +static inline int security_task_alloc(struct task_struct *p,
> > > + struct task_struct *task)
> > > {
> > >   return 0;
> > > }
> > > Index: upstream/kernel/fork.c
> > > ===
> > > --- upstream.orig/kernel/fork.c
> > > +++ upstream/kernel/fork.c
> > > @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
> > >   /* Perform scheduler related setup. Assign this task to a CPU. */
> > >   sched_fork(p, clone_flags);
> > >
> > > - if ((retval = security_task_alloc(p)))
> > > + if ((retval = security_task_alloc(p, task)))
> > >   goto bad_fork_cleanup_policy;
> > >   if ((retval = audit_alloc(p)))
> > >   goto bad_fork_cleanup_security;
> > > Index: upstream/security/dummy.c
> > > ===
> > > --- upstream.orig/security/dummy.c
> > > +++ upstream/security/dummy.c
> > > @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
> > >   return 0;
> > > }
> > >
> > > -static int dummy_task_alloc_security (struct task_struct *p)
> > > +static int dummy_task_alloc_security(struct task_struct *p,
> > > + struct task_struct *task)
> > > {
> > >   return 0;
> > > }
> > > Index: upstream/security/security.c
> > > ===
> > > --- upstream.orig/security/security.c
> > > +++ upstream/security/security.c
> > > @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
> > >   return security_ops->task_create(clone_flags);
> > > }
> > >
> > > -int security_task_alloc(struct task_struct *p)
> > > +int security_task_alloc(struct task_struct *p, struct task_struct
> > *task)
> > > {
> > > - return security_ops->task_alloc_security(p);
> > > + return security_ops->task_alloc_security(p, task);
> > > }
> > >

> > > void security_task_free(struct task_struct *p)
> > > Index: upstream/security/selinux/hooks.c
> > > ===
> > > --- upstream.orig/security/selinux/hooks.c
> > > +++ upstream/security/selinux/hooks.c
> > > @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
> > >   return task_has_perm(current, current, PROCESS__FORK);
> > > }
> > >
> > > -static int selinux_task_alloc_security(struct task_struct *tsk)
> > > +static int selinux_task_alloc_security(struct task_struct *tsk,
> > > +   struct task_struct *hijack_src)
> > > {
> > >   struct task_security_struct *tsec1, *tsec2;
> > >   int rc;
> > >
> > > + if (hijack_src != current)
> > > + return -EPERM;
> > > +
> > >   tsec1 = current->security;
> > >
> > >   rc = task_alloc_security(tsk);
> > > -
> > > To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> > > the body of a message to majordomo@vger.kernel.org
> > > More majordomo info at http://vger.kernel.org/majordomo-info.html
> > >
> > >
> >
> >
> > -
> > To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
> >
> >
> >
> >
> _____
> Message sent using UebiMiau 2.7.2

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Tue, 27 Nov 2007 16:01:27 GMT
View Forum Message <> Reply to Message

Quoting Casey Schaufler (casey@schaufler-ca.com):
>
> --- Mark Nelson <markn@au1.ibm.com> wrote:
>
> > Subject: [PATCH 2/2] hijack: update task_alloc_security
> >
> > Update task_alloc_security() to take the hijacked task as a second
> > argument.
>
> Could y'all bring me up to speed on what this is intended to
> accomplish so that I can understand the Smack implications?

It's basically like ptracing a process, forcing it to fork, then having
the child execute a file and continue as your child. It takes part of
its state from the current process (stack etc), some from the hijacked
process (namespaces, keys?), and an lsm can decide for itself whose ->security
should be used for the child process.

There are a few reasons to do this. One is to do an admin activity
inside a private namespace, i.e. doing a new mount in a vserver in which
only websphere is running.

Paul has another use for it - basically (iiuc) wanting to be able to
keep a set of namespaces around which he can always enter. For mounts
namespaces that can be accomplished independent of hijack by setting up
a private mount tree and pivot_root()ing into there. But for other
namespaces that isn't an option.

thanks,
-serge

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

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by serue on Tue, 27 Nov 2007 16:11:32 GMT
View Forum Message <> Reply to Message

Quoting Crispin Cowan (crispin@crispincowan.com):
> Just the name "sys_hijack" makes me concerned.
>
> This post describes a bunch of "what", but doesn't tell us about "why"

> we would want this. What is it for?

Please see my response to Casey's email.

> And I second Casey's concern about careful management of the privilege
> required to "hijack" a process.

Absolutely. We're definately still in RFC territory.

Note that there are currently several proposed (but no upstream) ways to
accomplish entering a namespace:

 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
 nsproxy is given an integer id. The id can be used to enter
 an nsproxy, basically a straight current->nsproxy = target_nsproxy;

 2. I had previously posted a patchset on top of the nsproxy
 cgroup which allowed entering a nsproxy through the ns cgroup
 interface.

There are objections to both those patchsets because simply switching a
task's nsproxy using a syscall or file write in the middle of running a
binary is quite unsafe. Eric Biederman had suggested using ptrace or
something like it to accomplish the goal.

Just using ptrace is however not safe either. You are inheriting *all*
of the target's context, so it shouldn't be difficult for a nefarious
container/vserver admin to trick the host admin into running something
which gives the container/vserver admin full access to the host.

That's where the hijack idea came from. Yes, I called it hijack to make
sure alarm bells went off :) bc it's definately still worrisome. But at
this point I believe it is the safest solution suggested so far.

-serge

> Crispin
>
> Mark Nelson wrote:
> > Here's the latest version of sys_hijack.
> > Apologies for its lateness.
> >
> > Thanks!
> >
> > Mark.
> >
> > Subject: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
> >

> > Move most of do_fork() into a new do_fork_task() which acts on
> > a new argument, task, rather than on current. do_fork() becomes
> > a call to do_fork_task(current, ...).
> >
> > Introduce sys_hijack (for i386 and s390 only so far). It is like
> > clone, but in place of a stack pointer (which is assumed null) it
> > accepts a pid. The process identified by that pid is the one
> > which is actually cloned. Some state - including the file
> > table, the signals and sighand (and hence tty), and the ->parent
> > are taken from the calling process.
> >
> > A process to be hijacked may be identified by process id, in the
> > case of HIJACK_PID. Alternatively, in the case of HIJACK_CG an
> > open fd for a cgroup 'tasks' file may be specified. The first
> > available task in that cgroup will then be hijacked.
> >
> > HIJACK_NS is implemented as a third hijack method. The main
> > purpose is to allow entering an empty cgroup without having
> > to keep a task alive in the target cgroup. When HIJACK_NS
> > is called, only the cgroup and nsproxy are copied from the
> > cgroup. Security, user, and rootfs info is not retained
> > in the cgroups and so cannot be copied to the child task.
> >
> > In order to hijack a process, the calling process must be
> > allowed to ptrace the target.
> >
> > Sending sigstop to the hijacked task can trick its parent shell
> > (if it is a shell foreground task) into thinking it should retake
> > its tty.
> >
> > So try not sending SIGSTOP, and instead hold the task_lock over
> > the hijacked task throughout the do_fork_task() operation.
> > This is really dangerous. I've fixed cgroup_fork() to not
> > task_lock(task) in the hijack case, but there may well be other
> > code called during fork which can under "some circumstances"
> > task_lock(task).
> >
> > Still, this is working for me.
> >
> > The effect is a sort of namespace enter. The following program
> > uses sys_hijack to 'enter' all namespaces of the specified task.
> > For instance in one terminal, do
> >
> > mount -t cgroup -ons cgroup /cgroup
> > hostname
> >   qemu
> > ns_exec -u /bin/sh
> >   hostname serge

> >   echo $$
> >   1073
> >   cat /proc/$$/cgroup
> >   ns:/node_1073
> >
> > In another terminal then do
> >
> > hostname
> >   qemu
> > cat /proc/$$/cgroup
> >   ns:/
> > hijack pid 1073
> >   hostname
> >   serge
> >   cat /proc/$$/cgroup
> >   ns:/node_1073
> > hijack cgroup /cgroup/node_1073/tasks
> >
> > Changelog:
> > Aug 23: send a stop signal to the hijacked process
> >   (like ptrace does).
> > Oct 09: Update for 2.6.23-rc8-mm2 (mainly pidns)
> >   Don't take task_lock under rcu_read_lock
> >   Send hijacked process to cgroup_fork() as
> >   the first argument.
> >   Removed some unneeded task_locks.
> > Oct 16: Fix bug introduced into alloc_pid.
> > Oct 16: Add 'int which' argument to sys_hijack to
> >   allow later expansion to use cgroup in place
> >   of pid to specify what to hijack.
> > Oct 24: Implement hijack by open cgroup file.
> > Nov 02: Switch copying of task info: do full copy
> >   from current, then copy relevant pieces from
> >   hijacked task.
> > Nov 06: Verbatim task_struct copy now comes from current,
> >   after which copy_hijackable_taskinfo() copies
> >   relevant context pieces from the hijack source.
> > Nov 07: Move arch-independent hijack code to kernel/fork.c
> > Nov 07: powerpc and x86_64 support (Mark Nelson)
> > Nov 07: Don't allow hijacking members of same session.
> > Nov 07: introduce cgroup_may_hijack, and may_hijack hook to
> >   cgroup subsystems. The ns subsystem uses this to
> >   enforce the rule that one may only hijack descendent
> >   namespaces.
> > Nov 07: s390 support
> > Nov 08: don't send SIGSTOP to hijack source task
> > Nov 10: cache reference to nsproxy in ns cgroup for use in
> >   hijacking an empty cgroup.

> > Nov 10: allow partial hijack of empty cgroup
> > Nov 13: don't double-get cgroup for hijack_ns
> >   find_css_set() actually returns the set with a
> >   reference already held, so cgroup_fork_fromcgroup()
> >   by doing a get_css_set() was getting a second
> >   reference. Therefore after exiting the hijack
> >   task we could not rmdir the csgroup.
> > Nov 22: temporarily remove x86_64 and powerpc support
> > Nov 27: rebased on 2.6.24-rc3
> >
> > ==
> > hijack.c
> > ==
> > /*
> > * Your options are:
> > * hijack pid 1078
> > * hijack cgroup /cgroup/node_1078/tasks
> > * hijack ns /cgroup/node_1078/tasks
> > */
> >
> > #define _BSD_SOURCE
> > #include <unistd.h>
> > #include <sys/syscall.h>
> > #include <sys/types.h>
> > #include <sys/wait.h>
> > #include <sys/stat.h>
> > #include <fcntl.h>
> > #include <sched.h>
> > #include <stdio.h>
> > #include <stdlib.h>
> > #include <string.h>
> >
> > #if __i386__
> > #   define __NR_hijack 325
> > #elif __s390x__
> > #   define __NR_hijack 319
> > #else
> > #   error "Architecture not supported"
> > #endif
> >
> > #ifndef CLONE_NEWUTS
> > #define CLONE_NEWUTS 0x04000000
> > #endif
> >
> > void usage(char *me)
> > {
> > printf("Usage: %s pid <pid>\n", me);
> > printf(" | %s cgroup <cgroup_tasks_file>\n", me);

```
> >  printf("     | %s ns <cgroup_tasks_file>\n", me);
> >  exit(1);
> > }
> >
> > int exec_shell(void)
> > {
> >  execl("/bin/sh", "/bin/sh", NULL);
> > }
> >
> > #define HIJACK_PID 1
> > #define HIJACK_CG 2
> > #define HIJACK_NS 3
> >
> > int main(int argc, char *argv[])
> > {
> >  int id;
> >  int ret;
> >  int status;
> >  int which_hijack;
> >
> >  if (argc < 3 || !strcmp(argv[1], "-h"))
> >   usage(argv[0]);
> >  if (strcmp(argv[1], "cgroup") == 0)
> >   which_hijack = HIJACK_CG;
> >  else if (strcmp(argv[1], "ns") == 0)
> >   which_hijack = HIJACK_NS;
> >  else
> >   which_hijack = HIJACK_PID;
> >
> >  switch(which_hijack) {
> >   case HIJACK_PID:
> >    id = atoi(argv[2]);
> >    printf("hijacking pid %d\n", id);
> >    break;
> >   case HIJACK_CG:
> >   case HIJACK_NS:
> >    id = open(argv[2], O_RDONLY);
> >    if (id == -1) {
> >     perror("cgroup open");
> >     return 1;
> >    }
> >    break;
> >  }
> >
> >  ret = syscall(__NR_hijack, SIGCHLD, which_hijack, (unsigned long)id);
> >
> >  if (which_hijack != HIJACK_PID)
> >   close(id);
```

```
> >  if  (ret == 0) {
> >   return exec_shell();
> >  } else if (ret < 0) {
> >   perror("sys_hijack");
> >  } else {
> >   printf("waiting on cloned process %d\n", ret);
> >   while(waitpid(-1, &status, __WALL) != -1)
> >     ;
> >   printf("cloned process exited with %d (waitpid ret %d)\n",
> >     status, ret);
> >  }
> >
> >  return ret;
> > }
> > ============================================================
> >
> > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > ---
> >  Documentation/cgroups.txt          |    9 +
> >  arch/s390/kernel/process.c         |   21 +++
> >  arch/x86/kernel/process_32.c       |   24 ++++
> >  arch/x86/kernel/syscall_table_32.S |    1
> >  include/asm-x86/unistd_32.h        |    3
> >  include/linux/cgroup.h             |   28 ++++-
> >  include/linux/nsproxy.h            |   12 +-
> >  include/linux/ptrace.h             |    1
> >  include/linux/sched.h              |   19 +++
> >  include/linux/syscalls.h           |    2
> >  kernel/cgroup.c                    |  133 ++++++++++++++++++++++-
> >  kernel/fork.c                      |  201 +++++++++++++++++++++++++++++++++---
> >  kernel/ns_cgroup.c                 |   88 +++++++++++++++-
> >  kernel/nsproxy.c                   |    4
> >  kernel/ptrace.c                    |    7 +
> >  15 files changed, 523 insertions(+), 30 deletions(-)
> >
> > Index: upstream/arch/s390/kernel/process.c
> > ============================================================
> > --- upstream.orig/arch/s390/kernel/process.c
> > +++ upstream/arch/s390/kernel/process.c
> > @@ -321,6 +321,27 @@ asmlinkage long sys_clone(void)
> >           parent_tidptr, child_tidptr);
> >  }
> >
> > +asmlinkage long sys_hijack(void)
> > +{
> > + struct pt_regs *regs = task_pt_regs(current);
> > + unsigned long sp = regs->orig_gpr2;
```

> > + unsigned long clone_flags = regs->gprs[3];
> > + int which = regs->gprs[4];
> > + unsigned int fd;
> > + pid_t pid;
> > +
> > + switch (which) {
> > + case HIJACK_PID:
> > + pid = regs->gprs[5];
> > + return hijack_pid(pid, clone_flags, *regs, sp);
> > + case HIJACK_CGROUP:
> > + fd = (unsigned int) regs->gprs[5];
> > + return hijack_cgroup(fd, clone_flags, *regs, sp);
> > + default:
> > + return -EINVAL;
> > + }
> > +}
> > +
> > /*
> >   * This is trivial, and on the face of it looks like it
> >   * could equally well be done in user mode.
> > Index: upstream/arch/x86/kernel/process_32.c
> > ===
> > --- upstream.orig/arch/x86/kernel/process_32.c
> > +++ upstream/arch/x86/kernel/process_32.c
> > @@ -37,6 +37,7 @@
> > #include <linux/personality.h>
> > #include <linux/tick.h>
> > #include <linux/percpu.h>
> > +#include <linux/cgroup.h>
> >
> > #include <asm/uaccess.h>
> > #include <asm/pgtable.h>
> > @@ -781,6 +782,29 @@ asmlinkage int sys_clone(struct pt_regs
> >   return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
> > }
> >
> > +asmlinkage int sys_hijack(struct pt_regs regs)
> > +{
> > + unsigned long sp = regs.esp;
> > + unsigned long clone_flags = regs.ebx;
> > + int which = regs.ecx;
> > + unsigned int fd;
> > + pid_t pid;
> > +
> > + switch (which) {
> > + case HIJACK_PID:
> > + pid = regs.edx;
> > + return hijack_pid(pid, clone_flags, regs, sp);

> > + case HIJACK_CGROUP:
> > + fd = (unsigned int) regs.edx;
> > + return hijack_cgroup(fd, clone_flags, regs, sp);
> > + case HIJACK_NS:
> > + fd = (unsigned int) regs.edx;
> > + return hijack_ns(fd, clone_flags, regs, sp);
> > + default:
> > + return -EINVAL;
> > + }
> > +}
> > +
> > /*
> >   * This is trivial, and on the face of it looks like it
> >   * could equally well be done in user mode.
> > Index: upstream/arch/x86/kernel/syscall_table_32.S
> > ===
> > --- upstream.orig/arch/x86/kernel/syscall_table_32.S
> > +++ upstream/arch/x86/kernel/syscall_table_32.S
> > @@ -324,3 +324,4 @@ ENTRY(sys_call_table)
> > .long sys_timerfd
> > .long sys_eventfd
> > .long sys_fallocate
> > + .long sys_hijack /* 325 */
> > Index: upstream/Documentation/cgroups.txt
> > ===
> > --- upstream.orig/Documentation/cgroups.txt
> > +++ upstream/Documentation/cgroups.txt
> > @@ -495,6 +495,15 @@ LL=cgroup_mutex
> > Called after the task has been attached to the cgroup, to allow any
> > post-attachment activity that requires memory allocations or blocking.
> >
> > +int may_hijack(struct cgroup_subsys *ss, struct cgroup *cont,
> > +   struct task_struct *task)
> > +LL=cgroup_mutex
> > +
> > +Called prior to hijacking a task. Current is cloning a new child
> > +which is hijacking cgroup, namespace, and security context from
> > +the target task. Called with the hijacked task locked. Return
> > +0 to allow.
> > +
> > void fork(struct cgroup_subsy *ss, struct task_struct *task)
> > LL=callback_mutex, maybe read_lock(tasklist_lock)
> >
> > Index: upstream/include/asm-x86/unistd_32.h
> > ===
> > --- upstream.orig/include/asm-x86/unistd_32.h
> > +++ upstream/include/asm-x86/unistd_32.h
> > @@ -330,10 +330,11 @@

> > #define __NR_timerfd 322
> > #define __NR_eventfd 323
> > #define __NR_fallocate 324
> > +#define __NR_hijack 325
> >
> > #ifdef __KERNEL__
> >
> > -#define NR_syscalls 325
> > +#define NR_syscalls 326
> >
> > #define __ARCH_WANT_IPC_PARSE_VERSION
> > #define __ARCH_WANT_OLD_READDIR
> > Index: upstream/include/linux/cgroup.h
> > ===
> > --- upstream.orig/include/linux/cgroup.h
> > +++ upstream/include/linux/cgroup.h
> > @@ -14,19 +14,23 @@
> > #include <linux/nodemask.h>
> > #include <linux/rcupdate.h>
> > #include <linux/cgroupstats.h>
> > +#include <linux/err.h>
> >
> > #ifdef CONFIG_CGROUPS
> >
> > struct cgroupfs_root;
> > struct cgroup_subsys;
> > struct inode;
> > +struct cgroup;
> >
> > extern int cgroup_init_early(void);
> > extern int cgroup_init(void);
> > extern void cgroup_init_smp(void);
> > extern void cgroup_lock(void);
> > extern void cgroup_unlock(void);
> > -extern void cgroup_fork(struct task_struct *p);
> > +extern void cgroup_fork(struct task_struct *parent, struct task_struct *p);
> > +extern void cgroup_fork_fromcgroup(struct cgroup *new_cg,
> > +   struct task_struct *child);
> > extern void cgroup_fork_callbacks(struct task_struct *p);
> > extern void cgroup_post_fork(struct task_struct *p);
> > extern void cgroup_exit(struct task_struct *p, int run_callbacks);
> > @@ -236,6 +240,8 @@ struct cgroup_subsys {
> >   void (*destroy)(struct cgroup_subsys *ss, struct cgroup *cont);
> >   int (*can_attach)(struct cgroup_subsys *ss,
> >   struct cgroup *cont, struct task_struct *tsk);
> > + int (*may_hijack)(struct cgroup_subsys *ss,
> > +   struct cgroup *cont, struct task_struct *tsk);
> >   void (*attach)(struct cgroup_subsys *ss, struct cgroup *cont,

```
> >     struct cgroup *old_cont, struct task_struct *tsk);
> >   void (*fork)(struct cgroup_subsys *ss, struct task_struct *task);
> > @@ -304,12 +310,21 @@ struct task_struct *cgroup_iter_next(str
> >       struct cgroup_iter *it);
> >  void cgroup_iter_end(struct cgroup *cont, struct cgroup_iter *it);
> >
> > +struct cgroup *cgroup_from_fd(unsigned int fd);
> > +struct task_struct *task_from_cgroup_fd(unsigned int fd);
> > +int cgroup_may_hijack(struct task_struct *tsk);
> >  #else /* !CONFIG_CGROUPS */
> > +struct cgroup {
> > +};
> >
> >  static inline int cgroup_init_early(void) { return 0; }
> >  static inline int cgroup_init(void) { return 0; }
> >  static inline void cgroup_init_smp(void) {}
> > -static inline void cgroup_fork(struct task_struct *p) {}
> > +static inline void cgroup_fork(struct task_struct *parent,
> > +      struct task_struct *p) {}
> > +static inline void cgroup_fork_fromcgroup(struct cgroup *new_cg,
> > +     struct task_struct *child) {}
> > +
> >  static inline void cgroup_fork_callbacks(struct task_struct *p) {}
> >  static inline void cgroup_post_fork(struct task_struct *p) {}
> >  static inline void cgroup_exit(struct task_struct *p, int callbacks) {}
> > @@ -322,6 +337,15 @@ static inline int cgroupstats_build(stru
> >   return -EINVAL;
> >  }
> >
> > +static inline struct cgroup *cgroup_from_fd(unsigned int fd) { return NULL; }
> > +static inline struct task_struct *task_from_cgroup_fd(unsigned int fd)
> > +{
> > + return ERR_PTR(-EINVAL);
> > +}
> > +static inline int cgroup_may_hijack(struct task_struct *tsk)
> > +{
> > + return 0;
> > +}
> >  #endif /* !CONFIG_CGROUPS */
> >
> >  #endif /* _LINUX_CGROUP_H */
> > Index: upstream/include/linux/nsproxy.h
> > ===================================================================
> > --- upstream.orig/include/linux/nsproxy.h
> > +++ upstream/include/linux/nsproxy.h
> > @@ -3,6 +3,7 @@
> >
> >  #include <linux/spinlock.h>
```

> > #include <linux/sched.h>
> > +#include <linux/err.h>
> >
> > struct mnt_namespace;
> > struct uts_namespace;
> > @@ -81,10 +82,17 @@ static inline void get_nsproxy(struct ns
> >   atomic_inc(&ns->count);
> > }
> >
> > +struct cgroup;
> > #ifdef CONFIG_CGROUP_NS
> > -int ns_cgroup_clone(struct task_struct *tsk);
> > +int ns_cgroup_clone(struct task_struct *tsk, struct nsproxy *nsproxy);
> > +int ns_cgroup_verify(struct cgroup *cgroup);
> > +void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup);
> > #else
> > -static inline int ns_cgroup_clone(struct task_struct *tsk) { return 0; }
> > +static inline int ns_cgroup_clone(struct task_struct *tsk,
> > + struct nsproxy *nsproxy) { return 0; }
> > +static inline int ns_cgroup_verify(struct cgroup *cgroup) { return 0; }
> > +static inline void copy_hijack_nsproxy(struct task_struct *tsk,
> > + struct cgroup *cgroup) {}
> > #endif
> >
> > #endif
> > Index: upstream/include/linux/ptrace.h
> > ===
> > --- upstream.orig/include/linux/ptrace.h
> > +++ upstream/include/linux/ptrace.h
> > @@ -97,6 +97,7 @@ extern void __ptrace_link(struct task_st
> > extern void __ptrace_unlink(struct task_struct *child);
> > extern void ptrace_untrace(struct task_struct *child);
> > extern int ptrace_may_attach(struct task_struct *task);
> > +extern int ptrace_may_attach_locked(struct task_struct *task);
> >
> > static inline void ptrace_link(struct task_struct *child,
> > struct task_struct *new_parent)
> > Index: upstream/include/linux/sched.h
> > ===
> > --- upstream.orig/include/linux/sched.h
> > +++ upstream/include/linux/sched.h
> > @@ -29,6 +29,13 @@
> > #define CLONE_NEWNET 0x40000000 /* New network namespace */
> >
> > /*
> > + * Hijack flags
> > + */
> > +#define HIJACK_PID 1 /* 'id' is a pid */

> > +#define HIJACK_CGROUP 2 /* 'id' is an open fd for a cgroup dir */
> > +#define HIJACK_NS 3 /* 'id' is an open fd for a cgroup dir */
> > +
> > +/*
> > * Scheduling policies
> > */
> > #define SCHED_NORMAL 0
> > @@ -1693,9 +1700,19 @@ extern int allow_signal(int);
> > extern int disallow_signal(int);
> >
> > extern int do_execve(char *, char __user * __user *, char __user * __user *, struct pt_regs *);
> > -extern long do_fork(unsigned long, unsigned long, struct pt_regs *, unsigned long, int __user
*, int __user *);
> > +extern long do_fork(unsigned long, unsigned long, struct pt_regs *,
> > + unsigned long, int __user *, int __user *);
> > struct task_struct *fork_idle(int);
> >
> > +extern int hijack_task(struct task_struct *task, unsigned long clone_flags,
> > + struct pt_regs regs, unsigned long sp);
> > +extern int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
> > + unsigned long sp);
> > +extern int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
> > + struct pt_regs regs, unsigned long sp);
> > +extern int hijack_ns(unsigned int fd, unsigned long clone_flags,
> > + struct pt_regs regs, unsigned long sp);
> > +
> > extern void set_task_comm(struct task_struct *tsk, char *from);
> > extern void get_task_comm(char *to, struct task_struct *tsk);
> >
> > Index: upstream/include/linux/syscalls.h
> > ===
> > --- upstream.orig/include/linux/syscalls.h
> > +++ upstream/include/linux/syscalls.h
> > @@ -614,4 +614,6 @@ asmlinkage long sys_fallocate(int fd, in
> >
> > int kernel_execve(const char *filename, char *const argv[], char *const envp[]);
> >
> > +asmlinkage long sys_hijack(unsigned long flags, int which, unsigned long id);
> > +
> > #endif
> > Index: upstream/kernel/cgroup.c
> > ===
> > --- upstream.orig/kernel/cgroup.c
> > +++ upstream/kernel/cgroup.c
> > @@ -44,6 +44,7 @@
> > #include <linux/kmod.h>
> > #include <linux/delayacct.h>
> > #include <linux/cgroupstats.h>

```
> > +#include <linux/file.h>
> >
> >  #include <asm/atomic.h>
> >
> > @@ -2442,15 +2443,25 @@ static struct file_operations proc_cgrou
> >   * At the point that cgroup_fork() is called, 'current' is the parent
> >   * task, and the passed argument 'child' points to the child task.
> >   */
> > -void cgroup_fork(struct task_struct *child)
> > +void cgroup_fork(struct task_struct *parent, struct task_struct *child)
> >  {
> > - task_lock(current);
> > - child->cgroups = current->cgroups;
> > + if (parent == current)
> > +  task_lock(parent);
> > + child->cgroups = parent->cgroups;
> >   get_css_set(child->cgroups);
> > - task_unlock(current);
> > + if (parent == current)
> > +  task_unlock(parent);
> >   INIT_LIST_HEAD(&child->cg_list);
> >  }
> >
> > +void cgroup_fork_fromcgroup(struct cgroup *new_cg, struct task_struct *child)
> > +{
> > + mutex_lock(&cgroup_mutex);
> > + child->cgroups = find_css_set(child->cgroups, new_cg);
> > + INIT_LIST_HEAD(&child->cg_list);
> > + mutex_unlock(&cgroup_mutex);
> > +}
> > +
> >  /**
> >   * cgroup_fork_callbacks - called on a new task very soon before
> >   * adding it to the tasklist. No need to take any locks since no-one
> > @@ -2801,3 +2812,117 @@ static void cgroup_release_agent(struct
> >   spin_unlock(&release_list_lock);
> >   mutex_unlock(&cgroup_mutex);
> >  }
> > +
> > +static inline int task_available(struct task_struct *task)
> > +{
> > + if (task == current)
> > +  return 0;
> > + if (task_session(task) == task_session(current))
> > +  return 0;
> > + switch (task->state) {
> > + case TASK_RUNNING:
> > + case TASK_INTERRUPTIBLE:
```

```
> > +  return 1;
> > + default:
> > +   return 0;
> > + }
> > +}
> > +
> > +struct cgroup *cgroup_from_fd(unsigned int fd)
> > +{
> > + struct file *file;
> > + struct cgroup *cgroup = NULL;;
> > +
> > + file = fget(fd);
> > + if (!file)
> > +  return NULL;
> > +
> > + if (!file->f_dentry || !file->f_dentry->d_sb)
> > +  goto out_fput;
> > + if (file->f_dentry->d_parent->d_sb->s_magic != CGROUP_SUPER_MAGIC)
> > +  goto out_fput;
> > + if (strcmp(file->f_dentry->d_name.name, "tasks"))
> > +  goto out_fput;
> > +
> > + cgroup = __d_cgrp(file->f_dentry->d_parent);
> > +
> > +out_fput:
> > + fput(file);
> > + return cgroup;
> > +}
> > +
> > +/*
> > + * Takes an integer which is a open fd in current for a valid
> > + * cgroupfs file.  Returns a task in that cgroup, with its
> > + * refcount bumped.
> > + * Since we have an open file on the cgroup tasks file, we
> > + * at least don't have to worry about the cgroup being freed
> > + * in the middle of this.
> > + */
> > +struct task_struct *task_from_cgroup_fd(unsigned int fd)
> > +{
> > + struct cgroup *cgroup;
> > + struct cgroup_iter it;
> > + struct task_struct *task = NULL;
> > +
> > + cgroup = cgroup_from_fd(fd);
> > + if (!cgroup)
> > +  return NULL;
> > +
> > + rcu_read_lock();
```

```
> > + cgroup_iter_start(cgroup, &it);
> > + do {
> > +  task = cgroup_iter_next(cgroup, &it);
> > +  if (task)
> > +   printk(KERN_NOTICE "task %d state %lx\n",
> > +    task->pid, task->state);
> > + } while (task && !task_available(task));
> > + cgroup_iter_end(cgroup, &it);
> > + if (task)
> > +  get_task_struct(task);
> > + rcu_read_unlock();
> > + return task;
> > +}
> > +
> > +/*
> > + * is current allowed to hijack tsk?
> > + * permission will also be denied elsewhere if
> > + *  current may not ptrace tsk
> > + *  security_task_alloc(new_task, tsk) returns -EPERM
> > + * Here we are only checking whether current may attach
> > + * to tsk's cgroup.  If you can't enter the cgroup, you can't
> > + * hijack it.
> > + *
> > + * XXX TODO This means that ns_cgroup.c will need to allow
> > + * entering all descendent cgroups, not just the immediate
> > + * child.
> > + */
> > +int cgroup_may_hijack(struct task_struct *tsk)
> > +{
> > + int ret = 0;
> > + struct cgroupfs_root *root;
> > +
> > + mutex_lock(&cgroup_mutex);
> > + for_each_root(root) {
> > +  struct cgroup_subsys *ss;
> > +  struct cgroup *cgroup;
> > +  int subsys_id;
> > +
> > +  /* Skip this hierarchy if it has no active subsystems */
> > +  if (!root->actual_subsys_bits)
> > +   continue;
> > +  get_first_subsys(&root->top_cgroup, NULL, &subsys_id);
> > +  cgroup = task_cgroup(tsk, subsys_id);
> > +  for_each_subsys(root, ss) {
> > +   if (ss->may_hijack) {
> > +    ret = ss->may_hijack(ss, cgroup, tsk);
> > +    if (ret)
> > +     goto out_unlock;
```

```
> > +   }
> > +  }
> > + }
> > +
> > +out_unlock:
> > + mutex_unlock(&cgroup_mutex);
> > + return ret;
> > +}
> > Index: upstream/kernel/fork.c
> > ===================================================================
> > --- upstream.orig/kernel/fork.c
> > +++ upstream/kernel/fork.c
> > @@ -189,7 +189,7 @@ static struct task_struct *dup_task_stru
> >    return NULL;
> >   }
> >
> > - setup_thread_stack(tsk, orig);
> > + setup_thread_stack(tsk, current);
> >
> >  #ifdef CONFIG_CC_STACKPROTECTOR
> >   tsk->stack_canary = get_random_int();
> > @@ -616,13 +616,14 @@ struct fs_struct *copy_fs_struct(struct
> >
> >  EXPORT_SYMBOL_GPL(copy_fs_struct);
> >
> > -static int copy_fs(unsigned long clone_flags, struct task_struct *tsk)
> > +static inline int copy_fs(unsigned long clone_flags,
> > +  struct task_struct *src, struct task_struct *tsk)
> >  {
> >   if (clone_flags & CLONE_FS) {
> > -  atomic_inc(&current->fs->count);
> > +  atomic_inc(&src->fs->count);
> >    return 0;
> >   }
> > - tsk->fs = __copy_fs_struct(current->fs);
> > + tsk->fs = __copy_fs_struct(src->fs);
> >   if (!tsk->fs)
> >    return -ENOMEM;
> >   return 0;
> > @@ -962,6 +963,42 @@ static void rt_mutex_init_task(struct ta
> >  #endif
> >  }
> >
> > +void copy_hijackable_taskinfo(struct task_struct *p,
> > +  struct task_struct *task)
> > +{
> > + p->uid = task->uid;
> > + p->euid = task->euid;
```

> > + p->suid = task->suid;
> > + p->fsuid = task->fsuid;
> > + p->gid = task->gid;
> > + p->egid = task->egid;
> > + p->sgid = task->sgid;
> > + p->fsgid = task->fsgid;
> > + p->cap_effective = task->cap_effective;
> > + p->cap_inheritable = task->cap_inheritable;
> > + p->cap_permitted = task->cap_permitted;
> > + p->keep_capabilities = task->keep_capabilities;
> > + p->user = task->user;
> > + /*
> > + * should keys come from parent or hijack-src?
> > + */
> > +#ifdef CONFIG_SYSVIPC
> > + p->sysvsem = task->sysvsem;
> > +#endif
> > + p->fs = task->fs;
> > + p->nsproxy = task->nsproxy;
> > +}
> > +
> > +#define HIJACK_SOURCE_TASK 1
> > +#define HIJACK_SOURCE_CG 2
> > +struct hijack_source_info {
> > + char type;
> > + union hijack_source_union {
> > + struct task_struct *task;
> > + struct cgroup *cgroup;
> > + } u;
> > +};
> > +
> > /*
> >   * This creates a new process as a copy of the old one,
> >   * but does not actually start it yet.
> > @@ -970,7 +1007,8 @@ static void rt_mutex_init_task(struct ta
> >   * parts of the process environment (as per the clone
> >   * flags). The actual kick-off is left to the caller.
> >   */
> > -static struct task_struct *copy_process(unsigned long clone_flags,
> > +static struct task_struct *copy_process(struct hijack_source_info *src,
> > +   unsigned long clone_flags,
> >   unsigned long stack_start,
> >   struct pt_regs *regs,
> >   unsigned long stack_size,
> > @@ -980,6 +1018,12 @@ static struct task_struct *copy_process(
> >   int retval;
> >   struct task_struct *p;
> >   int cgroup_callbacks_done = 0;

```
> > + struct task_struct *task;
> > +
> > + if (src->type == HIJACK_SOURCE_TASK)
> > +  task = src->u.task;
> > + else
> > +  task = current;
> >
> >   if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
> >    return ERR_PTR(-EINVAL);
> > @@ -1007,6 +1051,10 @@ static struct task_struct *copy_process(
> >   p = dup_task_struct(current);
> >   if (!p)
> >    goto fork_out;
> > + if (current != task)
> > +  copy_hijackable_taskinfo(p, task);
> > + else if (src->type == HIJACK_SOURCE_CG)
> > +  copy_hijack_nsproxy(p, src->u.cgroup);
> >
> >   rt_mutex_init_task(p);
> >
> > @@ -1084,7 +1132,10 @@ static struct task_struct *copy_process(
> >  #endif
> >   p->io_context = NULL;
> >   p->audit_context = NULL;
> > - cgroup_fork(p);
> > + if (src->type == HIJACK_SOURCE_CG)
> > +  cgroup_fork_fromcgroup(src->u.cgroup, p);
> > + else
> > +  cgroup_fork(task, p);
> >  #ifdef CONFIG_NUMA
> >    p->mempolicy = mpol_copy(p->mempolicy);
> >    if (IS_ERR(p->mempolicy)) {
> > @@ -1135,7 +1186,7 @@ static struct task_struct *copy_process(
> >    goto bad_fork_cleanup_audit;
> >   if ((retval = copy_files(clone_flags, p)))
> >    goto bad_fork_cleanup_semundo;
> > - if ((retval = copy_fs(clone_flags, p)))
> > + if ((retval = copy_fs(clone_flags, task, p)))
> >    goto bad_fork_cleanup_files;
> >   if ((retval = copy_sighand(clone_flags, p)))
> >    goto bad_fork_cleanup_fs;
> > @@ -1167,7 +1218,7 @@ static struct task_struct *copy_process(
> >   p->pid = pid_nr(pid);
> >   p->tgid = p->pid;
> >   if (clone_flags & CLONE_THREAD)
> > -  p->tgid = current->tgid;
> > +  p->tgid = task->tgid;
> >
```

```
> >   p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr : NULL;
> >   /*
> > @@ -1378,8 +1429,12 @@ struct task_struct * __cpuinit fork_idle
> >  {
> >   struct task_struct *task;
> >   struct pt_regs regs;
> > + struct hijack_source_info src;
> >
> > - task = copy_process(CLONE_VM, 0, idle_regs(&regs), 0, NULL,
> > + src.type = HIJACK_SOURCE_TASK;
> > + src.u.task = current;
> > +
> > + task = copy_process(&src, CLONE_VM, 0, idle_regs(&regs), 0, NULL,
> >       &init_struct_pid);
> >   if (!IS_ERR(task))
> >    init_idle(task, cpu);
> > @@ -1404,29 +1459,43 @@ static int fork_traceflag(unsigned clone
> >  }
> >
> >  /*
> > - *  Ok, this is the main fork-routine.
> > - *
> > - * It copies the process, and if successful kick-starts
> > - * it and waits for it to finish using the VM if required.
> > + * if called with task!=current, then caller must ensure that
> > + *    1. it has a reference to task
> > + *    2. current must have ptrace permission to task
> >   */
> > -long do_fork(unsigned long clone_flags,
> > +long do_fork_task(struct hijack_source_info *src,
> > +  unsigned long clone_flags,
> >         unsigned long stack_start,
> >         struct pt_regs *regs,
> >         unsigned long stack_size,
> >         int __user *parent_tidptr,
> >         int __user *child_tidptr)
> >  {
> > - struct task_struct *p;
> > + struct task_struct *p, *task;
> >   int trace = 0;
> >   long nr;
> >
> > + if (src->type == HIJACK_SOURCE_TASK)
> > +  task = src->u.task;
> > + else
> > +  task = current;
> > + if (task != current) {
> > +  /* sanity checks */
```

> > + /* we only want to allow hijacking the simplest cases */
> > + if (clone_flags & CLONE_SYSVSEM)
> > +   return -EINVAL;
> > + if (current->ptrace)
> > +   return -EPERM;
> > + if (task->ptrace)
> > +   return -EINVAL;
> > + }
> >   if (unlikely(current->ptrace)) {
> >   trace = fork_traceflag (clone_flags);
> >   if (trace)
> >   clone_flags |= CLONE_PTRACE;
> >   }
> >
> > - p = copy_process(clone_flags, stack_start, regs, stack_size,
> > + p = copy_process(src, clone_flags, stack_start, regs, stack_size,
> >   child_tidptr, NULL);
> >   /*
> >   * Do this prior waking up the new thread - the thread pointer
> > @@ -1484,6 +1553,106 @@ long do_fork(unsigned long clone_flags,
> >   return nr;
> > }
> >
> > +/*
> > + * Ok, this is the main fork-routine.
> > + *
> > + * It copies the process, and if successful kick-starts
> > + * it and waits for it to finish using the VM if required.
> > + */
> > +long do_fork(unsigned long clone_flags,
> > +   unsigned long stack_start,
> > +   struct pt_regs *regs,
> > +   unsigned long stack_size,
> > +   int __user *parent_tidptr,
> > +   int __user *child_tidptr)
> > +{
> > + struct hijack_source_info src = {
> > +   .type = HIJACK_SOURCE_TASK,
> > +   .u = { .task = current, },
> > + };
> > + return do_fork_task(&src, clone_flags, stack_start,
> > + regs, stack_size, parent_tidptr, child_tidptr);
> > +}
> > +
> > +/*
> > + * Called with task count bumped, drops task count before returning
> > + */
> > +int hijack_task(struct task_struct *task, unsigned long clone_flags,

```
> > +  struct pt_regs regs, unsigned long sp)
> > +{
> > + int ret = -EPERM;
> > + struct hijack_source_info src = {
> > +    .type = HIJACK_SOURCE_TASK,
> > +    .u = { .task = task, },
> > + };
> > +
> > + task_lock(task);
> > + put_task_struct(task);
> > + if (!ptrace_may_attach_locked(task))
> > +  goto out_unlock_task;
> > + if (task == current)
> > +  goto out_unlock_task;
> > + ret = cgroup_may_hijack(task);
> > + if (ret)
> > +  goto out_unlock_task;
> > + if (task->ptrace) {
> > +  ret = -EBUSY;
> > +  goto out_unlock_task;
> > + }
> > + ret = do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
> > +
> > +out_unlock_task:
> > + task_unlock(task);
> > + return ret;
> > +}
> > +
> > +int hijack_pid(pid_t pid, unsigned long clone_flags, struct pt_regs regs,
> > +        unsigned long sp)
> > +{
> > + struct task_struct *task;
> > +
> > + rcu_read_lock();
> > + task = find_task_by_vpid(pid);
> > + if (task)
> > +  get_task_struct(task);
> > + rcu_read_unlock();
> > +
> > + if (!task)
> > +  return -EINVAL;
> > +
> > + return hijack_task(task, clone_flags, regs, sp);
> > +}
> > +
> > +int hijack_cgroup(unsigned int fd, unsigned long clone_flags,
> > +    struct pt_regs regs, unsigned long sp)
> > +{
```

> > + struct task_struct *task;
> > +
> > + task = task_from_cgroup_fd(fd);
> > + if (!task)
> > + return -EINVAL;
> > +
> > + return hijack_task(task, clone_flags, regs, sp);
> > +}
> > +
> > +int hijack_ns(unsigned int fd, unsigned long clone_flags,
> > +   struct pt_regs regs, unsigned long sp)
> > +{
> > + struct hijack_source_info src;
> > + struct cgroup *cgroup;
> > +
> > + cgroup = cgroup_from_fd(fd);
> > + if (!cgroup)
> > + return -EINVAL;
> > +
> > + if (!ns_cgroup_verify(cgroup))
> > + return -EINVAL;
> > +
> > + src.type = HIJACK_SOURCE_CG;
> > + src.u.cgroup = cgroup;
> > + return do_fork_task(&src, clone_flags, sp, &regs, 0, NULL, NULL);
> > +}
> > +
> > #ifndef ARCH_MIN_MMSTRUCT_ALIGN
> > #define ARCH_MIN_MMSTRUCT_ALIGN 0
> > #endif
> > Index: upstream/kernel/ns_cgroup.c
> > ===
> > --- upstream.orig/kernel/ns_cgroup.c
> > +++ upstream/kernel/ns_cgroup.c
> > @@ -7,9 +7,11 @@
> > #include <linux/module.h>
> > #include <linux/cgroup.h>
> > #include <linux/fs.h>
> > +#include <linux/nsproxy.h>
> >
> > struct ns_cgroup {
> >   struct cgroup_subsys_state css;
> > + struct nsproxy *nsproxy;
> >   spinlock_t lock;
> > };
> >
> > @@ -22,9 +24,51 @@ static inline struct ns_cgroup *cgroup_t
> >   struct ns_cgroup, css);

```
> >  }
> >
> > -int ns_cgroup_clone(struct task_struct *task)
> > +int ns_cgroup_clone(struct task_struct *task, struct nsproxy *nsproxy)
> >  {
> > - return cgroup_clone(task, &ns_subsys);
> > + struct cgroup *cgroup;
> > + struct ns_cgroup *ns_cgroup;
> > + int ret = cgroup_clone(task, &ns_subsys);
> > +
> > + if (ret)
> > +  return ret;
> > +
> > + cgroup = task_cgroup(task, ns_subsys_id);
> > + ns_cgroup = cgroup_to_ns(cgroup);
> > + ns_cgroup->nsproxy = nsproxy;
> > + get_nsproxy(nsproxy);
> > +
> > + return 0;
> > +}
> > +
> > +int ns_cgroup_verify(struct cgroup *cgroup)
> > +{
> > + struct cgroup_subsys_state *css;
> > + struct ns_cgroup *ns_cgroup;
> > +
> > + css = cgroup_subsys_state(cgroup, ns_subsys_id);
> > + if (!css)
> > +  return 0;
> > + ns_cgroup = container_of(css, struct ns_cgroup, css);
> > + if (!ns_cgroup->nsproxy)
> > +  return 0;
> > + return 1;
> > +}
> > +
> > +/*
> > + * this shouldn't be called unless ns_cgroup_verify() has
> > + * confirmed that there is a ns_cgroup in this cgroup
> > + *
> > + * tsk is not yet running, and has not yet taken a reference
> > + * to it's previous ->nsproxy, so we just do a simple assignment
> > + * rather than switch_task_namespaces()
> > + */
> > +void copy_hijack_nsproxy(struct task_struct *tsk, struct cgroup *cgroup)
> > +{
> > + struct ns_cgroup *ns_cgroup;
> > +
> > + ns_cgroup = cgroup_to_ns(cgroup);
```

> > + tsk->nsproxy = ns_cgroup->nsproxy;
> > }
> >
> > /*
> > @@ -60,6 +104,42 @@ static int ns_can_attach(struct cgroup_s
> > return 0;
> > }
> >
> > +static void ns_attach(struct cgroup_subsys *ss,
> > + struct cgroup *cgroup, struct cgroup *oldcgroup,
> > + struct task_struct *tsk)
> > +{
> > + struct ns_cgroup *ns_cgroup = cgroup_to_ns(cgroup);
> > +
> > + if (likely(ns_cgroup->nsproxy))
> > + return;
> > +
> > + spin_lock(&ns_cgroup->lock);
> > + if (!ns_cgroup->nsproxy) {
> > + ns_cgroup->nsproxy = tsk->nsproxy;
> > + get_nsproxy(ns_cgroup->nsproxy);
> > + }
> > + spin_unlock(&ns_cgroup->lock);
> > +}
> > +
> > +/*
> > + * only allow hijacking child namespaces
> > + * Q: is it crucial to prevent hijacking a task in your same cgroup?
> > + */
> > +static int ns_may_hijack(struct cgroup_subsys *ss,
> > + struct cgroup *new_cgroup, struct task_struct *task)
> > +{
> > + if (current == task)
> > + return -EINVAL;
> > +
> > + if (!capable(CAP_SYS_ADMIN))
> > + return -EPERM;
> > +
> > + if (!cgroup_is_descendant(new_cgroup))
> > + return -EPERM;
> > +
> > + return 0;
> > +}
> > +
> > /*
> > * Rules: you can only create a cgroup if
> > * 1. you are capable(CAP_SYS_ADMIN)
> > @@ -88,12 +168,16 @@ static void ns_destroy(struct cgroup_sub

> >   struct ns_cgroup *ns_cgroup;
> >
> >   ns_cgroup = cgroup_to_ns(cgroup);
> > + if (ns_cgroup->nsproxy)
> > + put_nsproxy(ns_cgroup->nsproxy);
> >   kfree(ns_cgroup);
> > }
> >
> > struct cgroup_subsys ns_subsys = {
> >   .name = "ns",
> >   .can_attach = ns_can_attach,
> > + .attach = ns_attach,
> > + .may_hijack = ns_may_hijack,
> >   .create = ns_create,
> >   .destroy = ns_destroy,
> >   .subsys_id = ns_subsys_id,
> > Index: upstream/kernel/nsproxy.c
> > ===
> > --- upstream.orig/kernel/nsproxy.c
> > +++ upstream/kernel/nsproxy.c
> > @@ -144,7 +144,7 @@ int copy_namespaces(unsigned long flags,
> >   goto out;
> >   }
> >
> > - err = ns_cgroup_clone(tsk);
> > + err = ns_cgroup_clone(tsk, new_ns);
> >   if (err) {
> >   put_nsproxy(new_ns);
> >   goto out;
> > @@ -196,7 +196,7 @@ int unshare_nsproxy_namespaces(unsigned
> >   goto out;
> >   }
> >
> > - err = ns_cgroup_clone(current);
> > + err = ns_cgroup_clone(current, *new_nsp);
> >   if (err)
> >   put_nsproxy(*new_nsp);
> >
> > Index: upstream/kernel/ptrace.c
> > ===
> > --- upstream.orig/kernel/ptrace.c
> > +++ upstream/kernel/ptrace.c
> > @@ -159,6 +159,13 @@ int ptrace_may_attach(struct task_struct
> >   return !err;
> > }
> >
> > +int ptrace_may_attach_locked(struct task_struct *task)
> > +{

> > + int err;
> > + err = may_attach(task);
> > + return !err;
> > +}
> > +
> > int ptrace_attach(struct task_struct *task)
> > {
> >   int retval;
> > -
> > To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
>
> --
> Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
> CEO, Mercenary Linux http://mercenarylinux.com/
> Itanium. Vista. GPLv3. Complexity at work
>
> -
> To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Tue, 27 Nov 2007 17:05:02 GMT
View Forum Message <> Reply to Message

[resending bc the headers were screwed up in previous email, so
  two lists and one individual didn't get this. hopefully it was
  a user or transient error and goes through this time]

Quoting Rodrigo Rubira Branco (BSDaemon) (rodrigo@kernelhacking.com):
> It will give another easy way to locate selinux security structures inside
> the kernel, will not?

By locate, you mean actually finding the structures in kernel memory to
overwrite them?

> Again, if you have a kernel vulnerability and this feature, someone will
> easily disable selinux for the process, or just change the security concerns
> for it ;).

Maybe hijack should go under a kernel config for security reasons.

thanks,
-serge

> cya,
>
>
> Rodrigo (BSDaemon).
>
> --
> http://www.kernelhacking.com/rodrigo
>
> Kernel Hacking: If i really know, i can hack
>
> GPG KeyID: 1FCEDEA1
>
>
> --------- Mensagem Original --------
> De: Joshua Brindle <method@manicmethod.com>
> Para: Mark Nelson <markn@au1.ibm.com>
> C?pia: containers@lists.linux-foundation.org,
> linux-security-module@vger.kernel.org, selinux@tycho.nsa.gov,
> menage@google.com, Stephen Smalley <sds@tycho.nsa.gov>, James Morris
> <jmorris@namei.org>, Serge E. Hallyn <serue@us.ibm.com>
> Assunto: Re: [PATCH 2/2] hijack: update task_alloc_security
> Data: 27/11/07 02:38
>
> >
> > Mark Nelson wrote:
> > > Subject: [PATCH 2/2] hijack: update task_alloc_security
> > >
> > > Update task_alloc_security() to take the hijacked task as a second
> > > argument.
> > >
> > > For the selinux version, refuse permission if hijack_src!=current,
> > > since we have no idea what the proper behavior is. Even if we
> > > assume that the resulting child should be in the hijacked task's
> > > domain, depending on the policy that may not be enough information
> > > since init_t executing /bin/bash could result in a different domain
> > > than login_t executing /bin/bash.
> > >
> > >
> > This means its basically not possible to hijack tasks with SELinux
> > right? It would be a shame if this weren't useful to people running
> SELinux.
> >

> > It seems to me (I may be wrong, I'm sure someone will let me know if I
> > am) that the right way to handle this with SELinux is to check to see if
> > the current task (caller of sys_hijack) has permission to ptrace (or
> > some other permission deemed suitable, perhaps a new one) and if so copy
> > the security blob pointer from the hijacked task to the new one (we
> > don't want tranquility problems).
> >
> > From your paragraph above it seems like you were thinking there should
> > be a transition at hijack time but we don't automatically transition
> > anywhere except exec.
> >
> > Anyway, I just don't think you should completely disable this for
> > SELinux users.
> >
> > > Signed-off-by: Serge Hallyn <serue@us.ibm.com>
> > > Signed-off-by: Mark Nelson <markn@au1.ibm.com>
> > > ---
> > > include/linux/security.h | 12 ++++++++---
> > > kernel/fork.c | 2 +-
> > > security/dummy.c | 3 ++-
> > > security/security.c | 4 ++--
> > > security/selinux/hooks.c | 6 +++++-
> > > 5 files changed, 19 insertions(+), 8 deletions(-)
> > >
> > > Index: upstream/include/linux/security.h
> > > ===
> > > --- upstream.orig/include/linux/security.h
> > > +++ upstream/include/linux/security.h
> > > @@ -545,9 +545,13 @@ struct request_sock;
> > >   * Return 0 if permission is granted.
> > >   * @task_alloc_security:
> > >   * @p contains the task_struct for child process.
> > > + * @task contains the task_struct for process to be hijacked
> > >   * Allocate and attach a security structure to the p->security
> field. The
> > >   * security field is initialized to NULL when the task structure is
> > >   * allocated.
> > > + * @task will usually be current. If it is not equal to current,
> then
> > > + * a sys_hijack system call is going on, and current is asking for a
> > > + * child to be created in the context of the hijack src, @task.
> > >   * Return 0 if operation was successful.
> > >   * @task_free_security:
> > >   * @p contains the task_struct for process.
> > > @@ -1301,7 +1305,8 @@ struct security_operations {
> > >   int (*dentry_open) (struct file *file);
> > >
> > >   int (*task_create) (unsigned long clone_flags);

> > > - int (*task_alloc_security) (struct task_struct * p);
> > > + int (*task_alloc_security) (struct task_struct *p,
> > > + struct task_struct *task);
> > > void (*task_free_security) (struct task_struct * p);
> > > int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
> > > int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
> > > @@ -1549,7 +1554,7 @@ int security_file_send_sigiotask(struct
> > > int security_file_receive(struct file *file);
> > > int security_dentry_open(struct file *file);
> > > int security_task_create(unsigned long clone_flags);
> > > -int security_task_alloc(struct task_struct *p);
> > > +int security_task_alloc(struct task_struct *p, struct task_struct
> *task);
> > > void security_task_free(struct task_struct *p);
> > > int security_task_setuid(uid_t id0, uid_t id1, uid_t id2, int
> flags);
> > > int security_task_post_setuid(uid_t old_ruid, uid_t old_euid,
> > > @@ -2021,7 +2026,8 @@ static inline int security_task_create (
> > > return 0;
> > > }
> > >
> > > -static inline int security_task_alloc (struct task_struct *p)
> > > +static inline int security_task_alloc(struct task_struct *p,
> > > + struct task_struct *task)
> > > {
> > > return 0;
> > > }
> > > Index: upstream/kernel/fork.c
> > > ===
> > > --- upstream.orig/kernel/fork.c
> > > +++ upstream/kernel/fork.c
> > > @@ -1177,7 +1177,7 @@ static struct task_struct *copy_process(
> > > /* Perform scheduler related setup. Assign this task to a CPU. */
> > > sched_fork(p, clone_flags);
> > >
> > > - if ((retval = security_task_alloc(p)))
> > > + if ((retval = security_task_alloc(p, task)))
> > > goto bad_fork_cleanup_policy;
> > > if ((retval = audit_alloc(p)))
> > > goto bad_fork_cleanup_security;
> > > Index: upstream/security/dummy.c
> > > ===
> > > --- upstream.orig/security/dummy.c
> > > +++ upstream/security/dummy.c
> > > @@ -475,7 +475,8 @@ static int dummy_task_create (unsigned l
> > > return 0;
> > > }
> > >

> > > -static int dummy_task_alloc_security (struct task_struct *p)
> > > +static int dummy_task_alloc_security(struct task_struct *p,
> > > + struct task_struct *task)
> > > {
> > >   return 0;
> > > }
> > > Index: upstream/security/security.c
> > > ===
> > > --- upstream.orig/security/security.c
> > > +++ upstream/security/security.c
> > > @@ -568,9 +568,9 @@ int security_task_create(unsigned long c
> > >   return security_ops->task_create(clone_flags);
> > > }
> > >
> > > -int security_task_alloc(struct task_struct *p)
> > > +int security_task_alloc(struct task_struct *p, struct task_struct
> *task)
> > > {
> > > - return security_ops->task_alloc_security(p);
> > > + return security_ops->task_alloc_security(p, task);
> > > }
> > >
> > > void security_task_free(struct task_struct *p)
> > > Index: upstream/security/selinux/hooks.c
> > > ===
> > > --- upstream.orig/security/selinux/hooks.c
> > > +++ upstream/security/selinux/hooks.c
> > > @@ -2788,11 +2788,15 @@ static int selinux_task_create(unsigned
> > >   return task_has_perm(current, current, PROCESS__FORK);
> > > }
> > >
> > > -static int selinux_task_alloc_security(struct task_struct *tsk)
> > > +static int selinux_task_alloc_security(struct task_struct *tsk,
> > > + struct task_struct *hijack_src)
> > > {
> > >   struct task_security_struct *tsec1, *tsec2;
> > >   int rc;
> > >
> > > + if (hijack_src != current)
> > > + return -EPERM;
> > > +
> > >   tsec1 = current->security;
> > >
> > >   rc = task_alloc_security(tsk);
> > > -
> > > To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> > > the body of a message to majordomo@vger.kernel.org

> > > More majordomo info at http://vger.kernel.org/majordomo-info.html
> > >
> > >
> >
> >
> > -
> > To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
> >
> >
> >
>
> _____
> Message sent using UebiMiau 2.7.2

----- End forwarded message -----

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Stephen Smalley on Tue, 27 Nov 2007 18:09:24 GMT
View Forum Message <> Reply to Message

On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> Quoting Crispin Cowan (crispin@crispincowan.com):
> > Just the name "sys_hijack" makes me concerned.
> >
> > This post describes a bunch of "what", but doesn't tell us about "why"
> > we would want this. What is it for?
>
> Please see my response to Casey's email.
>
> > And I second Casey's concern about careful management of the privilege
> > required to "hijack" a process.
>
> Absolutely. We're definately still in RFC territory.
>
> Note that there are currently several proposed (but no upstream) ways to
> accomplish entering a namespace:
>

> 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> nsproxy is given an integer id. The id can be used to enter
> an nsproxy, basically a straight current->nsproxy = target_nsproxy;
>
> 2. I had previously posted a patchset on top of the nsproxy
> cgroup which allowed entering a nsproxy through the ns cgroup
> interface.
>
> There are objections to both those patchsets because simply switching a
> task's nsproxy using a syscall or file write in the middle of running a
> binary is quite unsafe. Eric Biederman had suggested using ptrace or
> something like it to accomplish the goal.
>
> Just using ptrace is however not safe either. You are inheriting *all*
> of the target's context, so it shouldn't be difficult for a nefarious
> container/vserver admin to trick the host admin into running something
> which gives the container/vserver admin full access to the host.

I don't follow the above - with ptrace, you are controlling a process
already within the container (hence in theory already limited to its
container), and it continues to execute within that container. What's
the issue there?

> That's where the hijack idea came from. Yes, I called it hijack to make
> sure alarm bells went off :) bc it's definately still worrisome. But at
> this point I believe it is the safest solution suggested so far.

--
Stephen Smalley
National Security Agency

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by serue on Tue, 27 Nov 2007 22:38:29 GMT
View Forum Message <> Reply to Message

Quoting Stephen Smalley (sds@tycho.nsa.gov):
> On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> > Quoting Crispin Cowan (crispin@crispincowan.com):
> > > Just the name "sys_hijack" makes me concerned.
> > >
> > > This post describes a bunch of "what", but doesn't tell us about "why"
> > > we would want this. What is it for?

> >
> > Please see my response to Casey's email.
> >
> > > And I second Casey's concern about careful management of the privilege
> > > required to "hijack" a process.
> >
> > Absolutely. We're definately still in RFC territory.
> >
> > Note that there are currently several proposed (but no upstream) ways to
> > accomplish entering a namespace:
> >
> > 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> > nsproxy is given an integer id. The id can be used to enter
> > an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> >
> > 2. I had previously posted a patchset on top of the nsproxy
> > cgroup which allowed entering a nsproxy through the ns cgroup
> > interface.
> >
> > There are objections to both those patchsets because simply switching a
> > task's nsproxy using a syscall or file write in the middle of running a
> > binary is quite unsafe. Eric Biederman had suggested using ptrace or
> > something like it to accomplish the goal.
> >
> > Just using ptrace is however not safe either. You are inheriting *all*
> > of the target's context, so it shouldn't be difficult for a nefarious
> > container/vserver admin to trick the host admin into running something
> > which gives the container/vserver admin full access to the host.
>
> I don't follow the above - with ptrace, you are controlling a process
> already within the container (hence in theory already limited to its
> container), and it continues to execute within that container. What's
> the issue there?

Hmm, yeah, I may have overspoken - I'm not good at making up exploits
but while I see it possible to confuse the host admin by setting bogus
environment, I guess there may not be an actual exploit.

Still after the fork induced through ptrace, we'll have to execute a
file out of the hijacked process' namespaces and path (unless we get
really 'exotic'). With hijack, execution continues under the caller's
control, which I do much prefer.

The remaining advantages of hijack over ptrace (beside "using ptrace for
that is crufty") are

 1. not subject to pid wraparound (when doing hijack_cgroup
    or hijack_ns)

2. ability to enter a namespace which has no active processes

These also highlight selinux issues. In the case of hijacking an
empty cgroup, there is no security context (because there is no task) so
the context of 'current' will be used. In the case of hijacking a
populated cgroup, a task is chosen "at random" to be the hijack source.

So there are two ways to look at deciding which context to use. Since
control continues in the original acting process' context, we might
want the child to continue in its context. However if the process
creates any objects in the virtual server, we don't want them
mislabeled, so we might want the task in the hijacked task's context.

Sigh. So here's why I thought I'd punt on selinux at least until I had
a working selinux-enforced container/vserver :)

-serge

PS: I'm certainly open to the suggestion that the kernel patch in the
end us as crufty as using ptrace.

> > That's where the hijack idea came from. Yes, I called it hijack to make
> > sure alarm bells went off :) bc it's definately still worrisome. But at
> > this point I believe it is the safest solution suggested so far.
>
> --
> Stephen Smalley
> National Security Agency
>
> -
> To unsubscribe from this list: send the line "unsubscribe linux-security-module" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Casey Schaufler on Tue, 27 Nov 2007 22:54:26 GMT
View Forum Message <> Reply to Message

--- "Serge E. Hallyn" <serue@us.ibm.com> wrote:

> Quoting Stephen Smalley (sds@tycho.nsa.gov):
> > On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> > > Quoting Crispin Cowan (crispin@crispincowan.com):

> > > > Just the name "sys_hijack" makes me concerned.
> > > >
> > > > This post describes a bunch of "what", but doesn't tell us about "why"
> > > > we would want this. What is it for?
> > >
> > > Please see my response to Casey's email.
> > >
> > > > And I second Casey's concern about careful management of the privilege
> > > > required to "hijack" a process.
> > >
> > > Absolutely. We're definately still in RFC territory.
> > >
> > > Note that there are currently several proposed (but no upstream) ways to
> > > accomplish entering a namespace:
> > >
> > > 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> > > nsproxy is given an integer id. The id can be used to enter
> > > an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> > >
> > > 2. I had previously posted a patchset on top of the nsproxy
> > > cgroup which allowed entering a nsproxy through the ns cgroup
> > > interface.
> > >
> > > There are objections to both those patchsets because simply switching a
> > > task's nsproxy using a syscall or file write in the middle of running a
> > > binary is quite unsafe. Eric Biederman had suggested using ptrace or
> > > something like it to accomplish the goal.
> > >
> > > Just using ptrace is however not safe either. You are inheriting *all*
> > > of the target's context, so it shouldn't be difficult for a nefarious
> > > container/vserver admin to trick the host admin into running something
> > > which gives the container/vserver admin full access to the host.
> >
> > I don't follow the above - with ptrace, you are controlling a process
> > already within the container (hence in theory already limited to its
> > container), and it continues to execute within that container. What's
> > the issue there?
>
> Hmm, yeah, I may have overspoken - I'm not good at making up exploits
> but while I see it possible to confuse the host admin by setting bogus
> environment, I guess there may not be an actual exploit.
>
> Still after the fork induced through ptrace, we'll have to execute a
> file out of the hijacked process' namespaces and path (unless we get
> *really* 'exotic'). With hijack, execution continues under the caller's
> control, which I do much prefer.
>
> The remaining advantages of hijack over ptrace (beside "using ptrace for

> that is crufty") are
>
> 1. not subject to pid wraparound (when doing hijack_cgroup
>   or hijack_ns)
> 2. ability to enter a namespace which has no active processes
>
> These also highlight selinux issues. In the case of hijacking an
> empty cgroup, there is no security context (because there is no task) so
> the context of 'current' will be used. In the case of hijacking a
> populated cgroup, a task is chosen "at random" to be the hijack source.
>
> So there are two ways to look at deciding which context to use. Since
> control continues in the original acting process' context, we might
> want the child to continue in its context. However if the process
> creates any objects in the virtual server, we don't want them
> mislabeled, so we might want the task in the hijacked task's context.

I wouldn't be surprised if you've been over this a dozen times
already, but why hijack an existing process instead of injecting
a new one with completely specified attributes? That way you don't
distinguish between an empty cgroup and a propulated one and you're
not at the mercy of the available hijackees. I know that I would be
much less uncomfortable with that schenario.

> Sigh. So here's why I thought I'd punt on selinux at least until I had
> a working selinux-enforced container/vserver :)
>
> -serge
>
> PS: I'm certainly open to the suggestion that the kernel patch in the
> end us as crufty as using ptrace.
>
> > > That's where the hijack idea came from. Yes, I called it hijack to make
> > > sure alarm bells went off :) bc it's definately still worrisome. But at
> > > this point I believe it is the safest solution suggested so far.
> > >
> > > --
> > > Stephen Smalley
> > > National Security Agency
> > >
> > > -
> > > To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in
> > > the body of a message to majordomo@vger.kernel.org
> > > More majordomo info at http://vger.kernel.org/majordomo-info.html
> -
> To unsubscribe from this list: send the line "unsubscribe
> linux-security-module" in

> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
>
>

Casey Schaufler
casey@schaufler-ca.com

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Crispin Cowan on Wed, 28 Nov 2007 05:50:00 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Stephen Smalley (sds@tycho.nsa.gov):
>
>> I agree with this part - we don't want people to have to choose between
>> using containers and using selinux, so if hijack is going to be a
>> requirement for effective use of containers, then we need to make them
>> work together.
>>
> Absolutely, we just need to decide how to properly make it work with
> selinux. Maybe we check for
>
> allow (current_domain):(hijacked_process_domain) hijack
> type_transition hijacked_process_domain \
>   vserver_enter_binary_t:process vserver1_hijack_admin_t;
>
Is there to be an LSM hook, so that modules can decide on an arbitrary
decision of whether to allow a hijack? So that this "do the right
SELinux" thing can be generalized for all LSMs to do the right thing.

Crispin

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
      Itanium. Vista. GPLv3. Complexity at work

_____
Containers mailing list

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Crispin Cowan on Wed, 28 Nov 2007 05:53:47 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Casey Schaufler (casey@schaufler-ca.com):
>
>> Could y'all bring me up to speed on what this is intended to
>> accomplish so that I can understand the Smack implications?
>>
> It's basically like ptracing a process, forcing it to fork, then having
> the child execute a file and continue as your child. It takes part of
> its state from the current process (stack etc), some from the hijacked
> process (namespaces, keys?), and an lsm can decide for itself whose ->security
> should be used for the child process.
>
That just doesn't gob smack me with the obvious abstract intention of
this API :)

So it is like I want to run a process inside a name space, but I am not
inside that name space, so I highjack one that is in there, force it to
fork, and then give me its child. Ugh.

Couldn't we just implement "put me in that namespace over there?" AFAIK
namespaces don't actually have names, making it hard to implement "put
me in namespace Foo", but I view that as a defect of namespaces that
should be fixed, rather than hacked around.

Crispin

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
  Itanium. Vista. GPLv3. Complexity at work

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

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)

Quoting Casey Schaufler (casey@schaufler-ca.com):
>
> --- "Serge E. Hallyn" <serue@us.ibm.com> wrote:
>
> > Quoting Stephen Smalley (sds@tycho.nsa.gov):
> > > On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> > > > Quoting Crispin Cowan (crispin@crispincowan.com):
> > > > > Just the name "sys_hijack" makes me concerned.
> > > > >
> > > > > This post describes a bunch of "what", but doesn't tell us about "why"
> > > > > we would want this. What is it for?
> > > >
> > > > Please see my response to Casey's email.
> > > >
> > > > > And I second Casey's concern about careful management of the privilege
> > > > > required to "hijack" a process.
> > > >
> > > > Absolutely. We're definately still in RFC territory.
> > > >
> > > > Note that there are currently several proposed (but no upstream) ways to
> > > > accomplish entering a namespace:
> > > >
> > > > 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> > > > nsproxy is given an integer id. The id can be used to enter
> > > > an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> > > >
> > > > 2. I had previously posted a patchset on top of the nsproxy
> > > > cgroup which allowed entering a nsproxy through the ns cgroup
> > > > interface.
> > > >
> > > > There are objections to both those patchsets because simply switching a
> > > > task's nsproxy using a syscall or file write in the middle of running a
> > > > binary is quite unsafe. Eric Biederman had suggested using ptrace or
> > > > something like it to accomplish the goal.
> > > >
> > > > Just using ptrace is however not safe either. You are inheriting *all*
> > > > of the target's context, so it shouldn't be difficult for a nefarious
> > > > container/vserver admin to trick the host admin into running something
> > > > which gives the container/vserver admin full access to the host.
> > >
> > > I don't follow the above - with ptrace, you are controlling a process
> > > already within the container (hence in theory already limited to its
> > > container), and it continues to execute within that container. What's
> > > the issue there?
> >
> > Hmm, yeah, I may have overspoken - I'm not good at making up exploits

> > but while I see it possible to confuse the host admin by setting bogus
> > environment, I guess there may not be an actual exploit.
> >
> > Still after the fork induced through ptrace, we'll have to execute a
> > file out of the hijacked process' namespaces and path (unless we get
> > *really* 'exotic'). With hijack, execution continues under the caller's
> > control, which I do much prefer.
> >
> > The remaining advantages of hijack over ptrace (beside "using ptrace for
> > that is crufty") are
> >
> > 1. not subject to pid wraparound (when doing hijack_cgroup
> > or hijack_ns)
> > 2. ability to enter a namespace which has no active processes
> >
> > These also highlight selinux issues. In the case of hijacking an
> > empty cgroup, there is no security context (because there is no task) so
> > the context of 'current' will be used. In the case of hijacking a
> > populated cgroup, a task is chosen "at random" to be the hijack source.
> >
> > So there are two ways to look at deciding which context to use. Since
> > control continues in the original acting process' context, we might
> > want the child to continue in its context. However if the process
> > creates any objects in the virtual server, we don't want them
> > mislabeled, so we might want the task in the hijacked task's context.
>
> I wouldn't be surprised if you've been over this a dozen times
> already, but why hijack an existing process instead of injecting
> a new one with completely specified attributes?

That's really all that hijack does. current is the one being cloned,
then we switch the namespace pointers over to those of the hijacked
process or cgroup. We also fix some ids which must be relative to the
task's new namespaces, i.e. pids, uids, etc. Clone is really the only
time we can sanely do this especially because of the pid namespace
construction.

Or, what do you mean by 'completely specified' attributes?

> That way you don't
> distinguish between an empty cgroup and a propulated one and you're
> not at the mercy of the available hijackees.

You aren't with hijack, because it is just a clone, and you continue
execution as though you'd just cloned. So mostly only namespace
pointers and some necessarily related resources are switched over.
Open files, stack, environment, they all come from current.

Looking back over copy_hijackable_info(), I suspect capabilities should
be kept from current, not taken from the hijacked task.

I'd like to say the same for the selinux context, but again then I worry
about files in the container being created with the wrong context.

> I know that I would be
> much less uncomfortable with that schenario.

I'm not clear on what your scenario is or why it would be more
comforting. Can you elaborate?

thanks,
-serge

> > Sigh. So here's why I thought I'd punt on selinux at least until I had
> > a working selinux-enforced container/vserver :)
> >
> > -serge
> >
> > PS: I'm certainly open to the suggestion that the kernel patch in the
> > end us as crufty as using ptrace.
> >
> > > > That's where the hijack idea came from. Yes, I called it hijack to make
> > > > sure alarm bells went off :) bc it's definately still worrisome. But at
> > > > this point I believe it is the safest solution suggested so far.
> > >
> > > --
> > > Stephen Smalley
> > > National Security Agency
> > >
> > > -
> > > To unsubscribe from this list: send the line "unsubscribe
> > linux-security-module" in
> > > the body of a message to majordomo@vger.kernel.org
> > > More majordomo info at http://vger.kernel.org/majordomo-info.html
> > -
> > To unsubscribe from this list: send the line "unsubscribe
> > linux-security-module" in
> > the body of a message to majordomo@vger.kernel.org
> > More majordomo info at http://vger.kernel.org/majordomo-info.html
> >
> >
> >
>
>
> Casey Schaufler
> casey@schaufler-ca.com

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Wed, 28 Nov 2007 14:54:22 GMT
View Forum Message <> Reply to Message

Quoting Crispin Cowan (crispin@crispincowan.com):
> Serge E. Hallyn wrote:
> > Quoting Stephen Smalley (sds@tycho.nsa.gov):
> >
> >> I agree with this part - we don't want people to have to choose between
> >> using containers and using selinux, so if hijack is going to be a
> >> requirement for effective use of containers, then we need to make them
> >> work together.
> >>
> > Absolutely, we just need to decide how to properly make it work with
> > selinux. Maybe we check for
> >
> > allow (current_domain):(hijacked_process_domain) hijack
> > type_transition hijacked_process_domain \
> >   vserver_enter_binary_t:process vserver1_hijack_admin_t;
> >
> Is there to be an LSM hook, so that modules can decide on an arbitrary
> decision of whether to allow a hijack? So that this "do the right
> SELinux" thing can be generalized for all LSMs to do the right thing.

Currently:

1. the permission is granted through ptrace
2. the lsm knows a hijack is going in security_task_alloc()
 when task != current

so the lsm has all the information it needs. But I have no objection
to a separate security_task_hijack() hook if you find the ptrace hook
insufficient.

-serge

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Wed, 28 Nov 2007 14:57:33 GMT
View Forum Message <> Reply to Message

Quoting Crispin Cowan (crispin@crispincowan.com):
> Serge E. Hallyn wrote:
> > Quoting Casey Schaufler (casey@schaufler-ca.com):
> >
> >> Could y'all bring me up to speed on what this is intended to
> >> accomplish so that I can understand the Smack implications?
> >>
> > It's basically like ptracing a process, forcing it to fork, then having
> > the child execute a file and continue as your child. It takes part of
> > its state from the current process (stack etc), some from the hijacked
> > process (namespaces, keys?), and an lsm can decide for itself whose ->security
> > should be used for the child process.
> >
> That just doesn't gob smack me with the obvious abstract intention of
> this API :)
>
> So it is like I want to run a process inside a name space, but I am not
> inside that name space, so I highjack one that is in there, force it to
> fork, and then give me its child. Ugh.

Well that's how it started, but in reality you are the one who forks,
and you only copy namespace and some related data.

So what you are objecting to is the process you'd have to go through if
we don't have hijack.

> Couldn't we just implement "put me in that namespace over there?" AFAIK

There are two patchsets out there to do that, and there are standing
objections to them. Really doing it at clone is the safest thing,
and with pid namespaces the only sane thing to do.

> namespaces don't actually have names, making it hard to implement "put
> me in namespace Foo", but I view that as a defect of namespaces that
> should be fixed, rather than hacked around.

You can name them using the ns cgroup.

thanks,
-serge

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

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Stephen Smalley on Wed, 28 Nov 2007 15:00:54 GMT
View Forum Message <> Reply to Message

On Tue, 2007-11-27 at 16:38 -0600, Serge E. Hallyn wrote:
> Quoting Stephen Smalley (sds@tycho.nsa.gov):
> > On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> > > Quoting Crispin Cowan (crispin@crispincowan.com):
> > > > Just the name "sys_hijack" makes me concerned.
> > > >
> > > > This post describes a bunch of "what", but doesn't tell us about "why"
> > > > we would want this. What is it for?
> > >
> > > Please see my response to Casey's email.
> > >
> > > > And I second Casey's concern about careful management of the privilege
> > > > required to "hijack" a process.
> > >
> > > Absolutely. We're definately still in RFC territory.
> > >
> > > Note that there are currently several proposed (but no upstream) ways to
> > > accomplish entering a namespace:
> > >
> > > 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> > > nsproxy is given an integer id. The id can be used to enter
> > > an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> > >
> > > 2. I had previously posted a patchset on top of the nsproxy
> > > cgroup which allowed entering a nsproxy through the ns cgroup
> > > interface.
> > >
> > > There are objections to both those patchsets because simply switching a
> > > task's nsproxy using a syscall or file write in the middle of running a
> > > binary is quite unsafe. Eric Biederman had suggested using ptrace or
> > > something like it to accomplish the goal.
> > >
> > > Just using ptrace is however not safe either. You are inheriting *all*
> > > of the target's context, so it shouldn't be difficult for a nefarious
> > > container/vserver admin to trick the host admin into running something
> > > which gives the container/vserver admin full access to the host.
> >
> > I don't follow the above - with ptrace, you are controlling a process
> > already within the container (hence in theory already limited to its
> > container), and it continues to execute within that container. What's
> > the issue there?
>
> Hmm, yeah, I may have overspoken - I'm not good at making up exploits
> but while I see it possible to confuse the host admin by setting bogus
> environment, I guess there may not be an actual exploit.

>
> Still after the fork induced through ptrace, we'll have to execute a
> file out of the hijacked process' namespaces and path (unless we get
> *really* 'exotic'). With hijack, execution continues under the caller's
> control, which I do much prefer.
>
> The remaining advantages of hijack over ptrace (beside "using ptrace for
> that is crufty") are
>
> 1. not subject to pid wraparound (when doing hijack_cgroup
> or hijack_ns)
> 2. ability to enter a namespace which has no active processes

So possibly I'm missing something, but the situation with hijack seems more exploitable than ptrace to me - you've created a hybrid task with one foot in current's world (open files, tty, connection to parent, executable) and one foot in the target's world (namespaces, uid/gid) which can then be leveraged by other tasks within the target's world/container as a way of breaking out of the container. No?

> These also highlight selinux issues. In the case of hijacking an
> empty cgroup, there is no security context (because there is no task) so
> the context of 'current' will be used. In the case of hijacking a
> populated cgroup, a task is chosen "at random" to be the hijack source.

Seems like you might be better off with a single operation for creating a new task within a given namespace set / cgroup rather than trying to handle multiple situations with different semantics / inheritance behavior. IOW, forget about hijacking a specific pid or picking a task at random from a populated cgroup - just always initialize the state of the newly created task in the same manner based solely on elements of the caller's state and the cgroup's state.

> So there are two ways to look at deciding which context to use. Since
> control continues in the original acting process' context, we might
> want the child to continue in its context. However if the process
> creates any objects in the virtual server, we don't want them
> mislabeled, so we might want the task in the hijacked task's context.

I suspect that we want to continue in the parent's context, and then the program can always use setfscreatecon() or exec a helper in a different context if it wants to create files with contexts tailored to the target.

> Sigh. So here's why I thought I'd punt on selinux at least until I had
> a working selinux-enforced container/vserver :)

--

Stephen Smalley
National Security Agency

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by serue on Wed, 28 Nov 2007 15:23:59 GMT
View Forum Message <> Reply to Message

Quoting Stephen Smalley (sds@tycho.nsa.gov):
> On Tue, 2007-11-27 at 16:38 -0600, Serge E. Hallyn wrote:
> > Quoting Stephen Smalley (sds@tycho.nsa.gov):
> > > On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> > > > Quoting Crispin Cowan (crispin@crispincowan.com):
> > > > > Just the name "sys_hijack" makes me concerned.
> > > > >
> > > > > This post describes a bunch of "what", but doesn't tell us about "why"
> > > > > we would want this. What is it for?
> > > > >
> > > > Please see my response to Casey's email.
> > > >
> > > > > And I second Casey's concern about careful management of the privilege
> > > > > required to "hijack" a process.
> > > > >
> > > > Absolutely. We're definately still in RFC territory.
> > > >
> > > > Note that there are currently several proposed (but no upstream) ways to
> > > > accomplish entering a namespace:
> > > >
> > > > 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> > > > nsproxy is given an integer id. The id can be used to enter
> > > > an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> > > >
> > > > 2. I had previously posted a patchset on top of the nsproxy
> > > > cgroup which allowed entering a nsproxy through the ns cgroup
> > > > interface.
> > > >
> > > > There are objections to both those patchsets because simply switching a
> > > > task's nsproxy using a syscall or file write in the middle of running a
> > > > binary is quite unsafe. Eric Biederman had suggested using ptrace or
> > > > something like it to accomplish the goal.
> > > >
> > > > Just using ptrace is however not safe either. You are inheriting *all*
> > > > of the target's context, so it shouldn't be difficult for a nefarious

> > > > container/vserver admin to trick the host admin into running something
> > > > which gives the container/vserver admin full access to the host.
> > >
> > > I don't follow the above - with ptrace, you are controlling a process
> > > already within the container (hence in theory already limited to its
> > > container), and it continues to execute within that container. What's
> > > the issue there?
> >
> > Hmm, yeah, I may have overspoken - I'm not good at making up exploits
> > but while I see it possible to confuse the host admin by setting bogus
> > environment, I guess there may not be an actual exploit.
> >
> > Still after the fork induced through ptrace, we'll have to execute a
> > file out of the hijacked process' namespaces and path (unless we get
> > *really* 'exotic'). With hijack, execution continues under the caller's
> > control, which I do much prefer.
> >
> > The remaining advantages of hijack over ptrace (beside "using ptrace for
> > that is crufty") are
> >
> > 1. not subject to pid wraparound (when doing hijack_cgroup
> >   or hijack_ns)
> > 2. ability to enter a namespace which has no active processes
>
> So possibly I'm missing something, but the situation with hijack seems
> more exploitable than ptrace to me - you've created a hybrid task with
> one foot in current's world (open files, tty, connection to parent,
> executable) and one foot in the target's world (namespaces, uid/gid)
> which can then be leveraged by other tasks within the target's
> world/container as a way of breaking out of the container. No?

I *think* the things coming out of the new container are well enough
chosen to prevent that. I see where you're opening up to being killed
by a task in the target container, though. But apart from setting a
PF_FLAG I'm not sure how to stop that anyway.

This actually reminds me that we need a valid uid in the target
namespace in the HIJACK_NS case. It's not a problem right now, but
as I was just looking at fixing up kernel/signal.c in light of user
namespaces, it is something to keep in mind.

> > These also highlight selinux issues. In the case of hijacking an
> > empty cgroup, there is no security context (because there is no task) so
> > the context of 'current' will be used. In the case of hijacking a
> > populated cgroup, a task is chosen "at random" to be the hijack source.
>
> Seems like you might be better off with a single operation for creating
> a new task within a given namespace set / cgroup rather than trying to

> handle multiple situations with different semantics / inheritance
> behavior. IOW, forget about hijacking a specific pid or picking a task
> at random from a populated cgroup - just always initialize the state of
> the newly created task in the same manner based solely on elements of
> the caller's state and the cgroup's state.

So you're saying implement only the HIJACK_NS?

I'm fine with that. Does anyone on the containers list object?

> > So there are two ways to look at deciding which context to use. Since
> > control continues in the original acting process' context, we might
> > want the child to continue in its context. However if the process
> > creates any objects in the virtual server, we don't want them
> > mislabeled, so we might want the task in the hijacked task's context.
>
> I suspect that we want to continue in the parent's context, and then the
> program can always use setfscreatecon() or exec a helper in a different
> context if it wants to create files with contexts tailored to the
> target.

That sounds good to me...

So we're looking at:

 1. drop HIJACK_PID and HIJACK_CGROUP

 2. have selinux_task_alloc_security() always set task->security
 to current->security and allow the hijack case.

thanks,
-serge

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Crispin Cowan on Thu, 29 Nov 2007 04:21:34 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Crispin Cowan (crispin@crispincowan.com):
>
>> Is there to be an LSM hook, so that modules can decide on an arbitrary
>> decision of whether to allow a hijack? So that this "do the right
>> SELinux" thing can be generalized for all LSMs to do the right thing.

>>
> Currently:
>
> 1. the permission is granted through ptrace
> 2. the lsm knows a hijack is going in security_task_alloc()
>   when task != current
>
> so the lsm has all the information it needs. But I have no objection
> to a separate security_task_hijack() hook if you find the ptrace hook
> insufficient.
>
I find that ptrace, specifically CAP_SYS_PTRACE, is overloaded. AppArmor
is having problems because we have to choose between granting
cap_sys_ptrace, or not allowing the process to read /proc/pid/self &
such like. So there, the problem is that we have to grant too much power
to a process to just let it read some /proc stuff about itself.

Here the problem appears to be the other way. cap_sys_ptrace is powerful
enough to mess with other user's processes on the system, but if ptrace
gives you hijack, then that seems to give you the power to control
processes in someone else's namespace.

Crispin

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
  Itanium. Vista. GPLv3. Complexity at work

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

---

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Thu, 29 Nov 2007 15:38:15 GMT
View Forum Message <> Reply to Message

Quoting Crispin Cowan (crispin@crispincowan.com):
> Serge E. Hallyn wrote:
> > Quoting Crispin Cowan (crispin@crispincowan.com):
> >
> >> Is there to be an LSM hook, so that modules can decide on an arbitrary
> >> decision of whether to allow a hijack? So that this "do the right
> >> SELinux" thing can be generalized for all LSMs to do the right thing.
> >>
> > Currently:

> >
> > 1. the permission is granted through ptrace
> > 2. the lsm knows a hijack is going in security_task_alloc()
> >   when task != current
> >
> > so the lsm has all the information it needs. But I have no objection
> > to a separate security_task_hijack() hook if you find the ptrace hook
> > insufficient.
> >
> I find that ptrace, specifically CAP_SYS_PTRACE, is overloaded. AppArmor
> is having problems because we have to choose between granting
> cap_sys_ptrace, or not allowing the process to read /proc/pid/self &
> such like. So there, the problem is that we have to grant too much power
> to a process to just let it read some /proc stuff about itself.
>
> Here the problem appears to be the other way. cap_sys_ptrace is powerful
> enough to mess with other user's processes on the system, but if ptrace
> gives you hijack, then that seems to give you the power to control
> processes in someone else's namespace.

The user namespace patchset I'm working on right now to start having
signals respect user namespaces introduces CAP_NS_OVERRIDE. Once that
is in, then hijack would require CAP_NS_OVERRIDE|CAP_SYS_PTRACE.

Of course, since we're considering only allowing HIJACK_NS which is
only allowed into a different namespace, hijack would then always
require CAP_NS_OVERRIDE...

Does that suffice?

If you still prefer an LSM hook, we can add that.

thanks,
-serge

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Mark Nelson on Fri, 30 Nov 2007 02:08:51 GMT
View Forum Message <> Reply to Message

Hi Paul and Eric,

Do you guys have any objections to dropping the hijack_pid() and
hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see

below for discussion)?

Thanks!

Mark.

Serge E. Hallyn wrote:
> Quoting Stephen Smalley (sds@tycho.nsa.gov):
>> On Tue, 2007-11-27 at 16:38 -0600, Serge E. Hallyn wrote:
>>> Quoting Stephen Smalley (sds@tycho.nsa.gov):
>>>> On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
>>>>> Quoting Crispin Cowan (crispin@crispincowan.com):
>>>>>> Just the name "sys_hijack" makes me concerned.
>>>>>>
>>>>>> This post describes a bunch of "what", but doesn't tell us about "why"
>>>>>> we would want this. What is it for?
>>>>> Please see my response to Casey's email.
>>>>>
>>>>>> And I second Casey's concern about careful management of the privilege
>>>>>> required to "hijack" a process.
>>>>> Absolutely. We're definately still in RFC territory.
>>>>>
>>>>> Note that there are currently several proposed (but no upstream) ways to
>>>>> accomplish entering a namespace:
>>>>>
>>>>> 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
>>>>> nsproxy is given an integer id. The id can be used to enter
>>>>> an nsproxy, basically a straight current->nsproxy = target_nsproxy;
>>>>>
>>>>> 2. I had previously posted a patchset on top of the nsproxy
>>>>> cgroup which allowed entering a nsproxy through the ns cgroup
>>>>> interface.
>>>>>
>>>>> There are objections to both those patchsets because simply switching a
>>>>> task's nsproxy using a syscall or file write in the middle of running a
>>>>> binary is quite unsafe. Eric Biederman had suggested using ptrace or
>>>>> something like it to accomplish the goal.
>>>>>
>>>>> Just using ptrace is however not safe either. You are inheriting *all*
>>>>> of the target's context, so it shouldn't be difficult for a nefarious
>>>>> container/vserver admin to trick the host admin into running something
>>>>> which gives the container/vserver admin full access to the host.
>>>> I don't follow the above - with ptrace, you are controlling a process
>>>> already within the container (hence in theory already limited to its
>>>> container), and it continues to execute within that container. What's
>>>> the issue there?
>>> Hmm, yeah, I may have overspoken - I'm not good at making up exploits

>>> but while I see it possible to confuse the host admin by setting bogus
>>> environment, I guess there may not be an actual exploit.
>>>
>>> Still after the fork induced through ptrace, we'll have to execute a
>>> file out of the hijacked process' namespaces and path (unless we get
>>> *really* 'exotic'). With hijack, execution continues under the caller's
>>> control, which I do much prefer.
>>>
>>> The remaining advantages of hijack over ptrace (beside "using ptrace for
>>> that is crufty") are
>>>
>>> 1. not subject to pid wraparound (when doing hijack_cgroup
>>>   or hijack_ns)
>>> 2. ability to enter a namespace which has no active processes
>> So possibly I'm missing something, but the situation with hijack seems
>> more exploitable than ptrace to me - you've created a hybrid task with
>> one foot in current's world (open files, tty, connection to parent,
>> executable) and one foot in the target's world (namespaces, uid/gid)
>> which can then be leveraged by other tasks within the target's
>> world/container as a way of breaking out of the container. No?
>
> I *think* the things coming out of the new container are well enough
> chosen to prevent that. I see where you're opening up to being killed
> by a task in the target container, though. But apart from setting a
> PF_FLAG I'm not sure how to stop that anyway.
>
> This actually reminds me that we need a valid uid in the target
> namespace in the HIJACK_NS case. It's not a problem right now, but
> as I was just looking at fixing up kernel/signal.c in light of user
> namespaces, it is something to keep in mind.
>
>>> These also highlight selinux issues. In the case of hijacking an
>>> empty cgroup, there is no security context (because there is no task) so
>>> the context of 'current' will be used. In the case of hijacking a
>>> populated cgroup, a task is chosen "at random" to be the hijack source.
>> Seems like you might be better off with a single operation for creating
>> a new task within a given namespace set / cgroup rather than trying to
>> handle multiple situations with different semantics / inheritance
>> behavior. IOW, forget about hijacking a specific pid or picking a task
>> at random from a populated cgroup - just always initialize the state of
>> the newly created task in the same manner based solely on elements of
>> the caller's state and the cgroup's state.
>
> So you're saying implement only the HIJACK_NS?
>
> I'm fine with that. Does anyone on the containers list object?
>
>>> So there are two ways to look at deciding which context to use. Since

>>> control continues in the original acting process' context, we might
>>> want the child to continue in its context. However if the process
>>> creates any objects in the virtual server, we don't want them
>>> mislabeled, so we might want the task in the hijacked task's context.
>> I suspect that we want to continue in the parent's context, and then the
>> program can always use setfscreatecon() or exec a helper in a different
>> context if it wants to create files with contexts tailored to the
>> target.
>
> That sounds good to me...
>
> So we're looking at:
>
> 1. drop HIJACK_PID and HIJACK_CGROUP
>
> 2. have selinux_task_alloc_security() always set task->security
> to current->security and allow the hijack case.
>
> thanks,
> -serge
>

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by Paul Menage on Fri, 30 Nov 2007 02:10:50 GMT
View Forum Message <> Reply to Message

On Nov 29, 2007 6:08 PM, Mark Nelson <markn@au1.ibm.com> wrote:
> Hi Paul and Eric,
>
> Do you guys have any objections to dropping the hijack_pid() and
> hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see
> below for discussion)?
>

hijack_ns() is the main bit that I care about anyway, so that's fine
by me. Are we planning on adding the other modes again later?

Paul

>
> Thanks!
>

> Mark.
>
>
> Serge E. Hallyn wrote:
> > Quoting Stephen Smalley (sds@tycho.nsa.gov):
> >> On Tue, 2007-11-27 at 16:38 -0600, Serge E. Hallyn wrote:
> >>> Quoting Stephen Smalley (sds@tycho.nsa.gov):
> >>>> On Tue, 2007-11-27 at 10:11 -0600, Serge E. Hallyn wrote:
> >>>>> Quoting Crispin Cowan (crispin@crispincowan.com):
> >>>>>> Just the name "sys_hijack" makes me concerned.
> >>>>>>
> >>>>>> This post describes a bunch of "what", but doesn't tell us about "why"
> >>>>>> we would want this. What is it for?
> >>>>> Please see my response to Casey's email.
> >>>>>
> >>>>>> And I second Casey's concern about careful management of the privilege
> >>>>>> required to "hijack" a process.
> >>>>> Absolutely. We're definately still in RFC territory.
> >>>>>
> >>>>> Note that there are currently several proposed (but no upstream) ways to
> >>>>> accomplish entering a namespace:
> >>>>>
> >>>>> 1. bind_ns() is a new pair of syscalls proposed by Cedric. An
> >>>>> nsproxy is given an integer id. The id can be used to enter
> >>>>> an nsproxy, basically a straight current->nsproxy = target_nsproxy;
> >>>>>
> >>>>> 2. I had previously posted a patchset on top of the nsproxy
> >>>>> cgroup which allowed entering a nsproxy through the ns cgroup
> >>>>> interface.
> >>>>>
> >>>>> There are objections to both those patchsets because simply switching a
> >>>>> task's nsproxy using a syscall or file write in the middle of running a
> >>>>> binary is quite unsafe. Eric Biederman had suggested using ptrace or
> >>>>> something like it to accomplish the goal.
> >>>>>
> >>>> Just using ptrace is however not safe either. You are inheriting *all*
> >>>> of the target's context, so it shouldn't be difficult for a nefarious
> >>>> container/vserver admin to trick the host admin into running something
> >>>> which gives the container/vserver admin full access to the host.
> >>> I don't follow the above - with ptrace, you are controlling a process
> >>> already within the container (hence in theory already limited to its
> >>> container), and it continues to execute within that container. What's
> >>> the issue there?
> >> Hmm, yeah, I may have overspoken - I'm not good at making up exploits
> >> but while I see it possible to confuse the host admin by setting bogus
> >> environment, I guess there may not be an actual exploit.
> >>
> >> Still after the fork induced through ptrace, we'll have to execute a

> >>> file out of the hijacked process' namespaces and path (unless we get
> >>> *really* 'exotic'). With hijack, execution continues under the caller's
> >>> control, which I do much prefer.
> >>>
> >>> The remaining advantages of hijack over ptrace (beside "using ptrace for
> >>> that is crufty") are
> >>>
> >>> 1. not subject to pid wraparound (when doing hijack_cgroup
> >>> or hijack_ns)
> >>> 2. ability to enter a namespace which has no active processes
> >> So possibly I'm missing something, but the situation with hijack seems
> >> more exploitable than ptrace to me - you've created a hybrid task with
> >> one foot in current's world (open files, tty, connection to parent,
> >> executable) and one foot in the target's world (namespaces, uid/gid)
> >> which can then be leveraged by other tasks within the target's
> >> world/container as a way of breaking out of the container. No?
> >
> > I *think* the things coming out of the new container are well enough
> > chosen to prevent that. I see where you're opening up to being killed
> > by a task in the target container, though. But apart from setting a
> > PF_FLAG I'm not sure how to stop that anyway.
> >
> > This actually reminds me that we need a valid uid in the target
> > namespace in the HIJACK_NS case. It's not a problem right now, but
> > as I was just looking at fixing up kernel/signal.c in light of user
> > namespaces, it is something to keep in mind.
> >
> >>> These also highlight selinux issues. In the case of hijacking an
> >>> empty cgroup, there is no security context (because there is no task) so
> >>> the context of 'current' will be used. In the case of hijacking a
> >>> populated cgroup, a task is chosen "at random" to be the hijack source.
> >> Seems like you might be better off with a single operation for creating
> >> a new task within a given namespace set / cgroup rather than trying to
> >> handle multiple situations with different semantics / inheritance
> >> behavior. IOW, forget about hijacking a specific pid or picking a task
> >> at random from a populated cgroup - just always initialize the state of
> >> the newly created task in the same manner based solely on elements of
> >> the caller's state and the cgroup's state.
> >
> > So you're saying implement only the HIJACK_NS?
> >
> > I'm fine with that. Does anyone on the containers list object?
> >
> >>> So there are two ways to look at deciding which context to use. Since
> >>> control continues in the original acting process' context, we might
> >>> want the child to continue in its context. However if the process
> >>> creates any objects in the virtual server, we don't want them
> >>> mislabeled, so we might want the task in the hijacked task's context.

> >> I suspect that we want to continue in the parent's context, and then the
> >> program can always use setfscreatecon() or exec a helper in a different
> >> context if it wants to create files with contexts tailored to the
> >> target.
> >
> > That sounds good to me...
> >
> > So we're looking at:
> >
> > 1. drop HIJACK_PID and HIJACK_CGROUP
> >
> > 2. have selinux_task_alloc_security() always set task->security
> > to current->security and allow the hijack case.
> >
> > thanks,
> > -serge
> >
>
>

_____

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by ebiederm on Fri, 30 Nov 2007 02:37:07 GMT
View Forum Message <> Reply to Message

Mark Nelson <markn@au1.ibm.com> writes:

> Hi Paul and Eric,
>
> Do you guys have any objections to dropping the hijack_pid() and
> hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see
> below for discussion)?

I need to step back and study what is being proposed.

My gut feeling is that you are proposing something that does not
support forking me a process inside a container so I can have a
shell without having to run a login program.

There is a reason I proposed ptrace as an initial prototype.

All of the other uses of enter in a namespace context I feel confident
we can support by just having proper virtual filesystems available
to processes outside of the container. For monitoring and control.

Eric

_____

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by serue on Fri, 30 Nov 2007 14:50:16 GMT
View Forum Message <> Reply to Message

Quoting Eric W. Biederman (ebiederm@xmission.com):
> Mark Nelson <markn@au1.ibm.com> writes:
>
> > Hi Paul and Eric,
> >
> > Do you guys have any objections to dropping the hijack_pid() and
> > hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see
> > below for discussion)?
>
> I need to step back and study what is being proposed.
>
> My gut feeling is that you are proposing something that does not
> support forking me a process inside a container so I can have a
> shell without having to run a login program.

Hmm, depends on exactly what you want, but you may be right.

In terms of namespaces it'll be in the target container, including
having a pid in the container.

The most dangerous part about the purely ptrace method you mention is
that pieces of the ptraced process' environment may leak, pollute,
and attack your new process. But it shouldn't be impossible to do
it safely. Just tedious.

> There is a reason I proposed ptrace as an initial prototype.
>
> All of the other uses of enter in a namespace context I feel confident
> we can support by just having proper virtual filesystems available
> to processes outside of the container. For monitoring and control.

I think you're showing an unhealthy amount of trust in both our ability
to provide full fs-based controls to all filesystems and to your own and
other people's abilities to never mess up a container. As an example of
the former, will you be able to create and configure a network interface
or add iptables rules purely through fs interface? As an example of the

latter, one little mistake and your container's mounts ns may no longer
be a slave of yours or of /containers/c_22/root. It might take you
years to figure out that all the time when you were doing

```
mount --bind /mnt/nas /containers/c_22/root/mnt/backup
echo 1 > /containers/c_22/root/root/backup-trigger
read /containers/c_22/root/root/backup-callback
umount /containers/c_22/root/mnt/backup
```

your backups weren't going to your network storage but just being copied
on local disk...

BUT more importantly, it sounds like you are not interested in
hijack_pid or hijack_cgroup, and Paul is only intersted in
hijack_ns. So noone will mind if we dump the other two? It
should greatly simplify the patch!

thanks,
-serge

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

---

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by serue on Fri, 30 Nov 2007 14:50:51 GMT
View Forum Message <> Reply to Message

Quoting Paul Menage (menage@google.com):
> On Nov 29, 2007 6:08 PM, Mark Nelson <markn@au1.ibm.com> wrote:
> > Hi Paul and Eric,
> >
> > Do you guys have any objections to dropping the hijack_pid() and
> > hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see
> > below for discussion)?
> >
>
> hijack_ns() is the main bit that I care about anyway, so that's fine
> by me. Are we planning on adding the other modes again later?

Don't think we are planning on it, but of course if people want it it
can always be revisited.

thanks,
-serge

_____
Containers mailing list

Subject: Re: [PATCH 1/2] namespaces: introduce sys_hijack (v10)
Posted by ebiederm on Fri, 30 Nov 2007 22:09:28 GMT
View Forum Message <> Reply to Message

"Serge E. Hallyn" <serue@us.ibm.com> writes:

> Quoting Eric W. Biederman (ebiederm@xmission.com):
>> Mark Nelson <markn@au1.ibm.com> writes:
>>
>> > Hi Paul and Eric,
>> >
>> > Do you guys have any objections to dropping the hijack_pid() and
>> > hijack_cgroup() parts of sys_hijack, leaving just hijack_ns() (see
>> > below for discussion)?
>>
>> I need to step back and study what is being proposed.
>>
>> My gut feeling is that you are proposing something that does not
>> support forking me a process inside a container so I can have a
>> shell without having to run a login program.
>
> Hmm, depends on exactly what you want, but you may be right.
>
> In terms of namespaces it'll be in the target container, including
> having a pid in the container.

Yes, which is generally what you want for a magic login shell.

> The most dangerous part about the purely ptrace method you mention is
> that pieces of the ptraced process' environment may leak, pollute,
> and attack your new process. But it shouldn't be impossible to do
> it safely. Just tedious.

Yes. It is that use case more then anything I am concerned with.

>> There is a reason I proposed ptrace as an initial prototype.
>>
>> All of the other uses of enter in a namespace context I feel confident
>> we can support by just having proper virtual filesystems available
>> to processes outside of the container. For monitoring and control.
>
> I think you're showing an unhealthy amount of trust in both our ability
> to provide full fs-based controls to all filesystems and to your own and

> other people's abilities to never mess up a container. As an example of
> the former, will you be able to create and configure a network interface
> or add iptables rules purely through fs interface?

Well the fs interface for monitoring is pretty much on target.
As for iptables just get me a proper socket outside of the container
and I can control things. (Pity we can't do plan 9 style binds of file
descriptors the mount namespace).

> As an example of the
> latter, one little mistake and your container's mounts ns may no longer
> be a slave of yours or of /containers/c_22/root. It might take you
> years to figure out that all the time when you were doing
>
> mount --bind /mnt/nas /containers/c_22/root/mnt/backup
> echo 1 > /containers/c_22/root/root/backup-trigger
> read /containers/c_22/root/root/backup-callback
> umount /containers/c_22/root/mnt/backup
>
> your backups weren't going to your network storage but just being copied
> on local disk...

Yes, that could be nasty.

> BUT more importantly, it sounds like you are not interested in
> hijack_pid or hijack_cgroup, and Paul is only intersted in
> hijack_ns. So noone will mind if we dump the other two? It
> should greatly simplify the patch!

I don't expect so. So far filesystem and file descriptor based
interfaces I am confident that we can use outside of a container
(which really is most of everything), with our current infrastructure.

Doing it that way seems to provide more natural access controls.

So I am mostly interested in some way to get a magic login shell
inside a chroot with a filedescriptor that I have passed for
my input and output. Make it a unix domain socket and I can
pass all of the filedescriptors I want in out of the little world.

I like the concept of using something like sys_hijack for that,
rather then ptrace, it can be a lot less of a hack.

I will come back to this and look a bit more once we have the pid
and network namespaces in decent shape. Thanks for keeping the
idea alive.

Eric

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Crispin Cowan on Sun, 02 Dec 2007 01:07:52 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Crispin Cowan (crispin@crispincowan.com):
>
>> I find that ptrace, specifically CAP_SYS_PTRACE, is overloaded. AppArmor
>> is having problems because we have to choose between granting
>> cap_sys_ptrace, or not allowing the process to read /proc/pid/self &
>> such like. So there, the problem is that we have to grant too much power
>> to a process to just let it read some /proc stuff about itself.
>>
>> Here the problem appears to be the other way. cap_sys_ptrace is powerful
>> enough to mess with other user's processes on the system, but if ptrace
>> gives you hijack, then that seems to give you the power to control
>> processes in someone else's namespace.
>>
> The user namespace patchset I'm working on right now to start having
> signals respect user namespaces introduces CAP_NS_OVERRIDE. Once that
> is in, then hijack would require CAP_NS_OVERRIDE|CAP_SYS_PTRACE.
>
> Of course, since we're considering only allowing HIJACK_NS which is
> only allowed into a different namespace, hijack would then always
> require CAP_NS_OVERRIDE...
>
> Does that suffice?
>
I think that CAP_NS_OVERRIDE|CAP_SYS_PTRACE is a problem because of the
| making ptrace more powerful than it is now. If you make it
CAP_NS_OVERRIDE only, then the problem goes away.

Crispin

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
  Itanium. Vista. GPLv3. Complexity at work

Containers mailing list

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by serue on Mon, 03 Dec 2007 14:50:12 GMT
View Forum Message <> Reply to Message

Quoting Crispin Cowan (crispin@crispincowan.com):
> Serge E. Hallyn wrote:
> > Quoting Crispin Cowan (crispin@crispincowan.com):
> >
> >> I find that ptrace, specifically CAP_SYS_PTRACE, is overloaded. AppArmor
> >> is having problems because we have to choose between granting
> >> cap_sys_ptrace, or not allowing the process to read /proc/pid/self &
> >> such like. So there, the problem is that we have to grant too much power
> >> to a process to just let it read some /proc stuff about itself.
> >>
> >> Here the problem appears to be the other way. cap_sys_ptrace is powerful
> >> enough to mess with other user's processes on the system, but if ptrace
> >> gives you hijack, then that seems to give you the power to control
> >> processes in someone else's namespace.
> >>
> > The user namespace patchset I'm working on right now to start having
> > signals respect user namespaces introduces CAP_NS_OVERRIDE. Once that
> > is in, then hijack would require CAP_NS_OVERRIDE|CAP_SYS_PTRACE.
> >
> > Of course, since we're considering only allowing HIJACK_NS which is
> > only allowed into a different namespace, hijack would then always
> > require CAP_NS_OVERRIDE...
> >
> > Does that suffice?
> >
> I think that CAP_NS_OVERRIDE|CAP_SYS_PTRACE is a problem because of the

Oops, yeah I meant &.

> | making ptrace more powerful than it is now. If you make it
> CAP_NS_OVERRIDE only, then the problem goes away.

I was seeing CAP_NS_OVERRIDE as more of a modifier to other
capabilities. So (CAP_NS_OVERRIDE&CAP_KILL) means you can
signal processes in another namespace.
(CAP_NS_OVERRIDE&CAP_DAC_OVERRIDE) means you can override (currently
nonexistent) DAC file access checks in another user namespace.
(CAP_NS_OVERRIDE&CAP_MKNOD) means you can create devices which your
container isn't allowed to create.

Yup, this is all somewhat looking ahead...

-serge

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

Subject: Re: [PATCH 2/2] hijack: update task_alloc_security
Posted by Crispin Cowan on Mon, 03 Dec 2007 19:43:23 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Crispin Cowan (crispin@crispincowan.com):
>
>> I think that CAP_NS_OVERRIDE|CAP_SYS_PTRACE is a problem because of the
> Oops, yeah I meant &.
>
Cool. With & then I have no problem at all.

Thanks,
Crispin

--
Crispin Cowan, Ph.D. http://crispincowan.com/~crispin
CEO, Mercenary Linux http://mercenarylinux.com/
Itanium. Vista. GPLv3. Complexity at work

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