# Creating a Secure Router with SELinux

Moving Information Protection
to the next Level

**Mike Anderson**
**Chief Scientist**
**The PTR Group, Inc.**
**mailto: mike@theptrgroup.com**
**http://www.theptrgroup.com**



## What We Will Talk About

- The Problem of Securing a Router/Firewall
- How does the U.S. Government view secure computing?
- What is SELinux?
- Layering security on an example device
  - We'll use a firewall/router
- Debugging the security policy
- Handling multiple security levels on the same machine
- Evaluations and the Common Criteria

## Router and Firewalls

- Very simply, a router is a device that handles packet transfer from one network to another
  - LAN to LAN, LAN to WAN/WAN to LAN, or between WAN segments
- Today, this service is typically combined with other capabilities such as NAT, DHCP and firewall features
  - The firewall feature is expected to provide a trusted bastion that allows for packet filtering
    - Helps keep the bad guys out of our networks

ELC-SFO-SELinux-3 

## Routers and "Feature Creep"

- Over the past few years, routers have become increasingly complex
  - Web browsers for configuration
  - SNMP for reporting
  - Use of IPTables for filtering
  - Addition of IPSEC
  - And much more…
- As we add new features, we add more code
  - This code likely has vulnerabilities

ELC-SFO-SELinux-4 

## Routers and Linux

- Increasingly, commercial routers are being implemented using Linux
  - Reasonably secure
  - Easily maintained
  - Already supports web browsers, IP filters, NAT, DHCP servers
- However, we know that Linux has security vulnerabilities
  - Not as bad as Windoze, thankfully ☺
  - But, still not up to handling highly sensitive data




## Discretionary Access Controls

- In Linux, we're most familiar with passwords and read/write/execute permissions
  - These are called Discretionary Access Controls (DAC)
- They're called discretionary because they are at a user's discretion to assign and employ them
  - There's no way for Linux to know who has the root password or protect against a hacked program

# Cranking up Security

- In order to ensure both confidentiality and integrity in a system, we need to be able to restrict both the behavior of applications and users
  - Preclude users from accessing applications and files they shouldn't
  - Constrain applications by enforcing a predefined behavior
    - Define a set of constraints in a security policy
- This level of security requires the employment of mandatory access controls (MAC)
  - Auditable actions that are not easily subverted

Source: pbskids.org




# The Principle of Least Privilege

- The foundation of traditional Government data security is that everything not explicitly allowed is denied
  - This is the principle of "least privilege"
- Users/applications are only allowed to do things that were foreseen in the security policy
  - No "I'll just become root to fix this" allowed
  - This is counter to the traditional Linux approach where everything is "flexible"
    - E.g., I'll use "cat" to create a configuration file

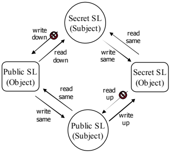
## Different Approaches to Security

- System-High Security
  - All subjects (programs, drivers, etc.) in the system have access to all objects (files, directories, sockets, etc.)
    - Typical RTOS
- Firewalled Security
  - Different system-high domains are separated by hardware/software that prevents sharing
    - Seen in many virtual machine/hypervisor approaches
- Transaction-Based Security
  - Each subject-object access is validated against a security policy
    - The approach of SELinux




## Confidentiality and Integrity

- Most believe that security implies confidentiality
  - Captured in the Bell-LaPadula (BLP) confidentiality model
    - "no read up, no write down"
- However, integrity is also important
  - Represented in the Biba integrity model
    - "no write up, no read down"
- A flexible security model must take both into account



# Security in the Linux Kernel

- Linux developers recognized the need for kernel-level security enforcement
  - They introduced the Linux Security Modules (LSM) framework into the 2.5/2.6 kernel development
- The LSM provides the hooks for alternate security models like LIDS, SELinux, AppArmor, etc.
- However, Linus did not feel that there was a security approach consensus for the kernel (circa 2001)
  - The National Security Agency (NSA) proposed SELinux as one approach
    - I.e, a worked example of how it could be done

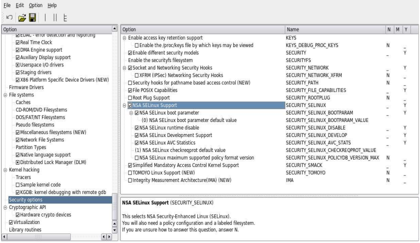



# LSM Hooks in the Kernel

- The LSM is implemented via a series of "hooks"
  - Your security model plugs in addresses for each of the hooks (security.h)

```
struct security_operations {
        int (*ptrace) (struct task_struct * parent, struct task_struct * child);
        int (*capget) (struct task_struct * target,
                       kernel_cap_t * effective,
                       kernel_cap_t * inheritable, kernel_cap_t * permitted);
        int (*capset_check) (struct task_struct * target,
                             kernel_cap_t * effective,
                             kernel_cap_t * inheritable,
                             kernel_cap_t * permitted);
        void (*capset_set) (struct task_struct * target,
                            kernel_cap_t * effective,
                            kernel_cap_t * inheritable,
                            kernel_cap_t * permitted);
        int (*capable) (struct task_struct * tsk, int cap);
        int (*acct) (struct file * file);
        int (*sysctl) (struct ctl_table * table, int op);
        int (*quotactl) (int cmds, int type, int id, struct super_block * sb);
        int (*quota_on) (struct dentry * dentry);
        int (*syslog) (int type);
        int (*settime) (struct timespec *ts, struct timezone *tz);
        int (*vm_enough_memory) (struct mm_struct *mm, long pages);
```

# Enabling SELinux in the Linux Kernel






# Example SELinux Hooks

- The security model then installs itself into the security hook structure (security/selinux/hooks.c)

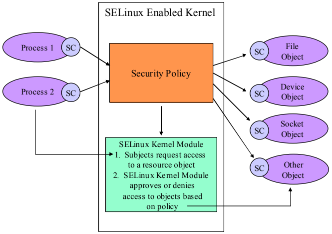
```
static struct security_operations selinux_ops = {
        .ptrace =                       selinux_ptrace,
        .capget =                       selinux_capget,
        .capset_check =                 selinux_capset_check,
        .capset_set =                   selinux_capset_set,
        .sysctl =                       selinux_sysctl,
        .capable =                      selinux_capable,
        .quotactl =                     selinux_quotactl,
        .quota_on =                     selinux_quota_on,
        .syslog =                       selinux_syslog,
        .vm_enough_memory =             selinux_vm_enough_memory,

        .netlink_send =                 selinux_netlink_send,
        .netlink_recv =                 selinux_netlink_recv,

        .bprm_alloc_security =          selinux_bprm_alloc_security,
        .bprm_free_security =           selinux_bprm_free_security,
        .bprm_apply_creds =             selinux_bprm_apply_creds,
        .bprm_post_apply_creds =        selinux_bprm_post_apply_creds,
        .bprm_set_security =            selinux_bprm_set_security,
        .bprm_check_security =          selinux_bprm_check_security,
        .bprm_secureexec =              selinux_bprm_secureexec,

        .sb_alloc_security =            selinux_sb_alloc_security,
        .sb_free_security =             selinux_sb_free_security,
        .sb_copy_data =                 selinux_sb_copy_data,
        .sb_kern_mount =                selinux_sb_kern_mount,
        .sb_statfs =                    selinux_sb_statfs,
        .sb_mount =                     selinux_mount,
        .sb_umount =                    selinux_umount,
```

## Security in the Kernel isn't Enough

- Enabling security in the kernel is a necessary, but insufficient step
  - We need security features in user space as well
- Essentially, we need to implement a defense-in-depth strategy
  - Assess the threat and implement features as needed
  - This means using both discretionary and mandatory access controls
    - And user-space libraries and applications to support them




## SELinux Architecture



## MAC via the LSM

- The use of the LSM allows the SELinux development team to implement a set of flexible MAC mechanisms in the kernel
  - Essentially, an implementation of NSA's "flask" security architecture
- The LSM hooks are integrated into the major kernel subsystems
  - No means to side-step the LSM
  - Provides for fine-grained object class and permission abstractions
- Each kernel object has a security context label associated with it
  - The use of the security context allows the kernel to enforce access decisions on kernel operations
- Security contexts have four security attributes
  - user:role:type:sensitivity label

ELC-SFO-SELinux-17 

## The SELinux Policy Engine

- Due to the NSA Flask legacy, the SELinux policy engine is referred to as the "security server"
- The policy engine implements:
  - Type Enforcement (TE) rules
  - Role-Based Access Control (RBAC) rules
  - Optional MLS/MCS separation
- The security policy is created via configuration files and then compiled and loaded into the security server
  - Ala kernel modules

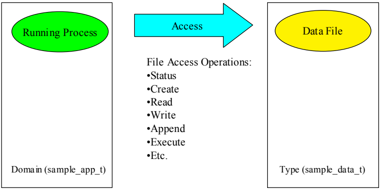
ELC-SFO-SELinux-18 

# Type-Enforcement Rules

- Creates “domains” for processes and types for objects
  - A domain is like a sand box
  - Think chroot jail on steroids
- Controls access to objects
  - Domain-to-type
- Controls process interactions
  - Domain-to-domain
- Controls entry into domains
  - Domain transitions
- Binds domains to executable code




# SELinux TE Diagram



# Example TE Rules

- Let apache create its PID file

```
allow apache_t var_run_t:dir {search add_name);
allow apache_t apache_var_run_t:file {create write}
type_transition apache_t var_run_t:file apache_var_run_t;
```

- Let VNC read its config file

```
allow vnc_t vnc_conf_t:file {getattr read);
```

- Let ssh bind a TCP socket

```
allow sshd_t ssh_port_t:tcp_socket name_bind;
```

- A complex system may have hundreds of thousands of TE rules
  - This screams for automated tools and macros




# Role-Based Access Control Rules

- Processes can be executed in a specific role
  - E.g., system admin, unprivileged user, etc.
- Limits which domains can be entered by each role
  - E.g., system admin can run "ifconfig" and "traceroute", but normal user can't
- Each user then has a set of authorized roles
- Sets a default domain for each user when they log in
- Uses TE rules to help manage the transitions and capabilities

## Sensitivity Labels

- The security context's last element is a sensitivity label
  - Comprised of a hierarchical sensitivity level and, optionally, one or more categories
    - Depending on the policy there can be 1 or 16 levels and 1024 categories
- The levels can be used for standard MLS applications
  - The categories can be viewed as "compartments"
  - Some commercial applications use the categories as successive access constraints




## Example Sensitivities

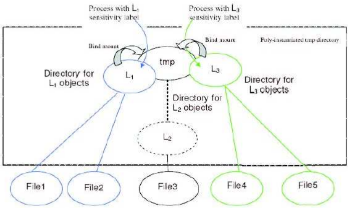
- s0:c0 is the lowest
- We can specify multiple categories at the same time
  - s0:c1,c10,c25
- Or ranges
  - s0:c6.c13
- The highest sensitivity level is
  - S15:c0.c1023
    - Also known as "System High"

## New File System Features

- The addition of MLS/MCS extensions also provides a means to segregate directories via “polyinstantiation”
- With polyinstantiation, each sensitivity level can see its own directory
  - An unclassified /tmp, secret /tmp, etc.
- Handled transparently by the O/S

## Polyinstantiated Directory Example



## File Contexts

- Each directory/file/dev node/symlink in the system also has additional security labeling information known as the file's context
- Example:

```
/usr/bin/appl - system_u:object_r:appl_t:s0:c0
```

- The file system must be labeled with the correct file contexts
  - The "fixfiles", "setfiles", and "restorecon" commands
- The file context then provides a mechanism to restrict access to each file system element by domain, user or role

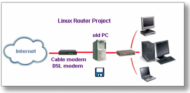



## Implementing the Router

- Given this SELinux background, we can now move on to the requirements to implement the router capability
- We next need to develop the requirements and security architecture document
  - What do we need the device to do?
  - What does it need to protect?
  - Are we MLS/MCS?
- This needs to be done in coordination with your sponsor organization

# Next Steps…

- Given the security architecture and requirements we can now start implementing something!
- We start with a good router design
  - Like the Linux router project




Source: pigtail.net

- Next, we enhance it with SELinux
  - This requires the definition of the security policies

# Security Engineering

- Given a router design, we need to isolate the IPCs
  - Who needs to talk to whom
  - Direction of the data flow
- We need to think in terms of uni-directional communications paths
  - Do not violate "no read down", etc.
  - Well-defined communications
- The SELinux sample "targeted" policy may be a good place to start
  - Allows everything but constrains only certain applications of concern
  - Progressively tighten the policy as you learn the interactions between applications
- However, security engineering is rarely a trivial effort
  - SELinux is not a panacea

# Warning: The BIOS is *Evil*


- Before we can create a device capable of handling secure information, we need to establish a root of trust within the device
  - Technically, this must start with the power-on jump to the BIOS and then move on to the boot loader
  - From there, we hit the O/S and the security policy
- Since we don't have control of the BIOS sources, we shouldn't trust them
  - CoreBoot, U-Boot or some other boot loader must be combined with a security device such as a Transaction Processing Module (TPM)
    - But, that's another talk altogether ☺

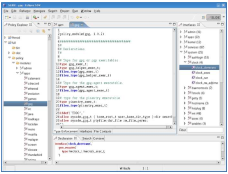



# Security Policy Life Cycle

- Policies are written as ASCII text files
  - Specialized IDEs such as the SLIDE Eclipse plug-in, Polgen or SEEdit can be used to ease policy creation
    - I did my first policy in "vi" ☺
- The policy is then checked for syntactic correctness using the "checkpolicy" command
- Next, you compile the policy using "make"
  - This produces a policy binary or a loadable policy module
- Finally, you load the policy using "load_policy"
  - Test, test, test…

## Example Policy Tool: SLIDE

- Built as an Eclipse Plug-in
- Allows editing the policy as well as compiling it for inclusion to the kernel
- Just one of many tools for SELinux that have been developed



Source: oss.tresys.com




## Testing a New Policy

- We can use the “setenforce” command to switch between strict and permissive mode
  - Permissive mode logs a violation but doesn’t deny the access
- Access vector (AV) information is then logged to /var/log/messages
  - Tools like “audit2allow” and “audit2why” help figure out what is happening

# Sample Logfile Entry

- Here is an example of the AVC logging output

```
Jan 18 19:56:08 localhost kernel:
  audit(1087602968.172.0): avc: denied {read }
  for pid=16577 exe=/usr/bin/tail name=messages dev=sda2
  ino=618992
  scontext=root:staff_r:staff_t
  tcontext=system_u:object_r:var_log_t
  tclass=file
```

What was denied?

What was accessed and by what program?

Source context?

Target context?




# The Policy-writer's Friend: -Z

- Many of the key Linux user commands have been enhanced to support the -Z option
  - Shows security context
- ls, ps, dir, find, install, mkdir, killall, pstree, stat, vdir and sudo/sudoedit all have support for -Z
- Given a log entry, we can use the -Z options to examine the security contexts that are causing the failures

## Modifying the Policy

- Once we have the log file entries:
  - We then deduce which “allows” or role transitions are needed to address the failure
  - Next, we modify the policy
  - Then, rebuild the policy and reload it
  - Finally, try the access again to see if the change solved the problem
- Debugging the policy is an iterative and rather time consuming process
- Next, we need to be evaluated…
  - This requires an outside evaluation organization

ELC-SFO-SELinux-37 

## Evaluation

- The old Orange Book has been superseded by the Common Criteria (CC) (ISO/IEC 15408)
  - An international standard for computer security
- The CC consists of a series of protection profiles
  - CAPP, LSPP, RBACPP
    - These are now technically retired and have been replaced with “Robustness” level protection profiles
- The device is then evaluated to an Evaluation Assurance Level (EAL 1–7)
  - See http://en.wikipedia.org/wiki/Evaluation_Assurance_Level for a quick overview of the EALs

ELC-SFO-SELinux-38 

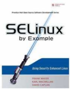
## SELinux and the CC

- RHEL 5/5.1 and SLES 10 were successfully evaluated at EAL 4+
- This includes the Common Access Protection Profile (CAPP)
  - Equivalent to the old Orange Book C1 level
- RHEL 5.1 also added Labeled Security PP (LSPP) and Role-Based Access Control PP (RBACPP)
  - Roughly equivalent to the Orange Book B1/B2 level
  - Also added network packet security labeling
    - "secmark"

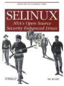


## Summary

- SELinux adds significant additional hardening
  - Used in conjunction with IPTables, IPSEC labeling, etc. and other "good security practices"
  - Subsystems like "tripwire" can be used as well
- Develop the device's requirements and security architecture
- Limit the number of applications and their files
- Develop the security policy and test it thoroughly
- Submit for evaluation if needed



Source: amazon.com



Source: amazon.com