Usable High-Assurance Operating Systems

Doug McIlroy
Sean Smith
Sergey Bratus
Alex Ferguson

Motivation

- Available high-assurance systems have large monolithic policies
  - Hard to write and maintain
  - Crafted by trial and error
  - E.g.: SELinux
    - FC3 “strict” policy.conf has 227275 lines
    - Trial-and-error prevalent in tutorials
      - Enshrined in **audit2allow** tool – run program repeatedly, record triggered perms, make record trail into policy
- Goal: explore the structured approach to security policy engineering

Expected Benefits

- More succinct policies
  - Easier to understand and create
- Higher confidence
  - Easier to audit
  - Less trial-and-error
- More amenable to automated security assessment and verification

SELinux observations (1)

- Fedora Core 3 “strict” policy:
  - Some structure is provided by M4 macros
    - 2000+ lines in the core macro base
      - domain_auto_trans, file_type_auto_trans
      - daemon_core_rules, uses_shlibs, ...
  - “Strict” policy defines 160+ domains
    - Each domain definition is a set of M4 macros
      - 24 lines (arpwatch) to 350 (postfix), 70 lines average
    - Some use core macros, some do not
      - It is too easy to jump levels – good style is not enforced
- Very hard to check in macro-preserving way

SELinux observations (2)

Example of policy statements:

```
type  sshd_key_t,   file_type,   sysadmfile;
...
allow  sysadm_t  sysadmfile:file
        { create ioctl read write append unlink rename };
type  sysadm_ssh_t,   domain,   privlog;
...
# if  something can log to syslog it should be able to log to the console
allow  privlog  console_device_t:chr_file
        { ioctl read write getattr };
...
type_transition  sysadm_ssh_t  tmp_t:{ file lnk_file
          sock_file fifo_file } sshd_tmp_t;
```

Sample SELinux macros

```
define(`create_file_perms'  `,{ create ioctl read getattr lock write
     setattr append link unlink rename }')
# can_ptrace(domain, domain)
define(`can_ptrace' `,
    allow  $1  $2:process ptrace;
    allow  $2  $1:process sigchld;
')
define(`daemon_core_rules'  `,
    type  $1_t,  domain,  privlog  $2;
    type  $1_exec_t,  file_type,  sysadmfile,  exec_type;
    role  system_r  types $1_t;
    # Inherit and use descriptors from init.
    allow  $1_t  init_t:fd  use;
    allow  $1_t  init_t:process  sigchld;
    allow  $1_t  self:process { signal_perms  fork };
    uses_shlib($1_t)
   ....
```

SELinux observations (3)

- Based on Linux Security Modules arch
  - 145 hooks in syscalls (~SELinux *operations*)
  - Linux DAC permission checking logic affected design (MAC check follows existing DAC checks, if DAC succeeds)
    - `permission(struct inode* , ...)` calls
      `generic_permission( inode )` -- DAC -- then
      `security_inode_permission` -- LSM --
- Implications of this on policies?

Observations summarized

- One policy language and mechanism is expected to address many goals:
  - Access control
  - Integrity
  - Confidentiality
  - Inevitable administrative exceptions, delegation
  - Hence, adequate expressive power for some tasks, but not for others, policy bloat
- SELinux/LSM were designed to express all kinds of possible MACs
  - ... and the goto operator could express all kinds of program topologies (Doug)
- Structure imposed by M4 macros is *implicit*

The IX approach

- IX (McIlroy, Reeds) is an early high-assurance version of UNIX
  - Supports lattice-policy MAC in UNIX
- MAC exceptions are described as privileges
  - There are only 6 “orthogonal” privileges
    - E.g.: lower a security label on a file (“declassify document”)
    - Assign an initial label to a data stream after authentication to a foreign host
- Privileges are administered outside of the MAC model, by the privserver
  - Highly structured
  - About ½ of all added code (3000 lines of C)

The IX privserver

- Handles admin tasks and delegation
  - Breaks this functionality out of MAC
- Describes “deviations” from MAC
  - Hierarchical, structured mechanism
  - Conceptually separate from MAC, in language and formalism
  - Specifies which commands can be performed by what actors with what parameters (in /etc/privs)
  - 150 lines to describe most common administrative tasks on a UNIX workstation on a small campus

IX privs formalism (1)

/etc/privs describes the priv tree, with nodes identified by Unix-like pathnames

Nodes have rights; a child node has a subset of its parent's rights; the root has all rights:

```
RIGHTS / operator, secadmin, downgrade(latticetop),...
RIGHTS /admin   operator, ...
RIGHTS /secure  secadmin, ...
RIGHTS /state                  downgrade(alldeskbits)
RIGHTS /state/desk/iraq        downgrade(iraqbits)
RIGHTS /state/desk/iran        downgrade(iranbits)
```

Rules govern access to priv-tree nodes:

```
ACCESS  /state  ID(condi)
ACCESS  /state  SRC(/dev/console), PW(daypassword)
```

IX privs formalism (2)

REQUEST rules govern admissibility and conduct of privileged actions. A request may NEED multiple rights. It may be executed provided the requestor has access to a priv-tree node that has those rights.

Example of a request to exercise privilege:

```
priv declassify qaedabits deltareconreport
```

```
REQUEST(declassify)   # clear some bits from file label
   NEEDS  downgrade($1)
   DOES   PRIV(nx),   # set up necessary privileges
          EXEC(setlab -s $1 $2) # subtract label $1
                                #  from file $2
```

Setlab is a small program that checks whether it can read or write the file, then tries to change the file's label. When run without privilege it obeys the MAC policy that labels can't decrease.

Project plans

- Apply the IX privserver model to SELinux
  - Give the admins a means of expressing a structured policy, concisely
  - Study its usability and effectiveness
  - Separate and compactify various parts of the SELinux policy
- Consider other high-assurance systems?

More about IX

http://www.cs.dartmouth.edu/~doug/IX/

Thank you

- Questions?

Title

- Item