4.5. Policy Languages

A **policy language** is a language for representing a security policy. High-level policy languages express policy constraints on entities using abstractions. Low-level policy languages express constraints in terms of input or invocation options to programs existing on the systems.

4.5.1. High-Level Policy Languages

A policy is independent of the mechanisms. It describes constraints placed on entities and actions in a system. A high-level policy language is an unambiguous expression of policy. Such precision requires a mathematical or programmatic formulation of policy; common English is not precise enough.

Assume that a system is connected to the Internet. A user runs a World Wide Web browser. Web browsers download programs from remote sites and execute them locally. The local system's policy may constrain what these downloaded programs can do.

EXAMPLE: Java is a programming language designed for programs to be downloaded by Web browsers. Pandey and Hashii [794] developed a policy constraint language for Java programs. Their high-level policy language specifies access constraints on resources and on how those constraints are inherited.

Their language expresses entities as classes and methods. A **class** is a set of objects to which a particular access constraint may be applied; a **method** is the set of ways in which an operation can be invoked. **Instantiation** occurs when a subject **s** creates an instance of a class **c**, and is written **s** -| **c**. **Invocation** occurs when a subject $\mathbf{s}_1$ executes an object $\mathbf{s}_2$ (which becomes a subject, because it is active) and is written $\mathbf{s}_1 \mid\!\rightarrow \mathbf{s}_2$. A **condition** is a Boolean condition. Access constraints are of the form

```
deny(s op x) when b
```

where **op** is -| or |→, **s** is a subject, **x** is another subject or a class, and **b** is a Boolean expression. This constraint states that subject **s** cannot perform operation **op** on **x** when condition **b** is true. If **s** is omitted, the action is forbidden to all entities.

Inheritance causes access constraints to be conjoined. Specifically, let class $\mathbf{c}_1$ define a method **f**, and have a subclass $\mathbf{c}_2$. Then $\mathbf{c}_2$ inherits **f**. Assume that the constraints are

```
deny(s -| c1.f) when
b1
```

```
deny(s -| c₂.f) when
b₂
```

A subclass inherits constraints on the parent class. Hence, **both** constraints **b_1** and **b_2** constrain **c_2**'s invocation of **f**. The appropriate constraint is

```
deny(s -| c₂.f) when
b₁ ∨ b₂
```

Suppose the policy states that the downloaded program is not allowed to access the password file on a UNIX system. The program accesses local files using the following class and methods.

```
class File {
public file(String name);
public String getfilename();
public char read();
...
```

Then the appropriate constraint would be

```
deny( |→ file.read) when (file.getfilename() ==
"/etc/passwd")
```

As another example, let the class **Socket** define the network interface, and let the method **Network.numconns** define the number of network connections currently active. The following access constraint bars any new connections when 100 connections are currently open.

```
deny( -| Socket) when (Network.numconns >= 100).
```

This language ignores implementation issues, and so is a high-level policy language. The **domain-type enforcement language** (DTEL) [54] grew from an observation of Boebert and Kain [126] that access could be based on types; they confine their work to the types “data” and “instructions.” This observation served as the basis for a firewall [996] and for other secure system components. DTEL uses implementation-level constructs to express constraints in terms of language types, but not as arguments or input to specific system commands. Hence, it combines elements of low-level and high-level languages. Because it describes configurations in the abstract, it is a high-level policy language.

EXAMPLE: DTEL associates a type with each object and a domain with each subject. The constructs of the language constrain the actions that a member of a domain can perform on an object of a specific type. For example, a subject cannot execute a text file, but it can execute an object file.

Consider a policy that restricts all users from writing to system binaries. Only subjects in the administrative domain can alter system binaries. A user can enter this domain only after rigorous authentication checks. In the UNIX world, this suggests four distinct subject domains:

1. **d_user**, the domain for ordinary users
2. **d_admin**, the domain for administrative users (who can alter system binaries)
3. **d_login**, the domain for the authentication processes that comply with the domain-type enforcement
4. **d_daemon**, the domain for system daemons (including those that spawn login)

The **login** program (in the **d_login** domain) controls access between **d_user** and **d_admin**. The system begins in the **d_daemon** domain because the **init** process lies there (and **init** spawns the **login** process whenever anyone tries to log in).

The policy suggests five object types:

1. **t_sysbin**, the type for executable files
2. **t_readable**, the type for readable files
3. **t_writable**, the type for writable files
4. **t_dte**, the type for data used by the domain-type enforcement mechanisms
5. **t_generic**, for data generated from user processes

For our purposes, consider these partitions. (In practice, objects can be both readable and writable; we ignore this for purposes of exposition.) DTEL represents this as

```
type t_readable, t_writable, t_sysbin, t_dte, t_generic;
```

Characterize each domain as a sequence. The first component is a list of the programs that begin in this domain. The other elements of the sequence consist of a set of rights, an arrow, and a type. Each element describes the set of rights that members of the domain have over the named type.

EXAMPLE: Consider the **d_daemon** domain. When the **init** program begins, it starts in this domain. It can create (c), read (r), write (w), and do a directory search (d) of any object of type **t_writable**. It can read, search, and execute (x) any object of type **t_sysbin**. It can read and search anything of type **t_generic**, **t_readable**, or **t_dte**. Finally, when the **init** program invokes the **login** program, the **login** program transitions into the **d_login** domain automatically. Putting this together, we have

```
domain d_daemon =  (/sbin/init),
                   (crwd->t_writable),
                   (rd->t_generic,t_readable, t_dte),
                   (rxd->t_sysbin),
                   (auto->d_login);
```

An important observation is that, even if a subject in the domain **d_daemon** is compromised, the attacker cannot alter system binaries (files of type **t_sysbin**), because that domain has no write rights over files of that type. This implements separation of privilege (see Section 13.2.6) and was a motive behind the development of this policy.

EXAMPLE: As a second example, the policy requires that only administrative subjects (domain **d_admin**) be able to write system executables (of type **t_sysbin**). The administrator uses an ordinary UNIX command interpreter. Subjects in **d_admin** can create objects of types **t_readable**, **t_writable**, **t_dte**, and **t_sysbin** (because they can write to those types) and can read, write, execute, and search any object of those types. If the type is not specified at creation, the new object is assigned the **t_generic** type. Finally, a subject in this domain can suspend processes executing in the **d_daemon** domain using the **sigtstp** signal. This means

```
domain d_admin = (/usr/bin/sh, /usr/bin/csh, /usr/bin/ksh),
                 (crwxd->t_generic),
                 (crwxd->t_readable, t_writable, t_dte,
                         t_sysbin),
                 (sigtstp->d_daemon);
```

The user domain must be constrained similarly. Here, users can write only objects of type **t_writable**, can execute only objects of type **t_sysbin**, can create only objects of type **t_writable** or **t_generic**, and can read and search all domains named.

```
domain d_user =  (/usr/bin/sh, /usr/bin/csh, /usr/bin/ksh),
                 (crwxd->t_generic),
                 (rxd->t_sysbin),
                 (crwd->t_writable),
                 (rd->t_readable, t_dte);
```

Because no user commands imply a transition out of the domain, the final component is empty.

The **d_login** domain controls access to the **d_user** and **d_admin** domains. Because this is its **only** function, no subject in that domain should be able to execute another program. It also is authorized to change the user ID (hence, it has the right **setauth**). Access to the domain is to be restricted to the **login** program. In other words,

```
domain d_login =  (/usr/bin/login),
                  (crwd->t_writable),
                  (rd->t_readable, t_generic, t_dte),
                  setauth,
                  (exec->d_user, d_admin);
```

Initially, the system starts in the **d_daemon** state:

```
initial_domain = d_daemon;
```

A series of assign statements sets the initial types of objects. For example,

```
assign −r t_generic /;
assign −r t_writable /usr/var, /dev, /tmp;
assign −r t_readable /etc;
assign −r −s dte_t /dte;
assign −r −s t_sysbin /sbin, /bin, /usr/bin, /usr/sbin
```

The −r flag means that the type is applied recursively; the −s flag means that the type is bound to the name, so if the object is deleted and a new object is created with the same name, the new object will have the same type as the deleted object. The **assign** lines are processed in order, so everything on the system without a type assigned by the last four lines is of type **t_generic** (because of the first line).

If a user process tries to alter a system binary, the enforcement mechanisms will check to determine if something in the domain **d_user** is authorized to write to an object of type **t_sysbin**. Because the domain description does not allow this, the request is refused.

Now augment the policy above to prevent users from modifying system logs. Define a new type **t_log** for the log files. Only subjects in the **d_admin** domain, and in a new domain **d_log**, can alter the log files. The set of domains would be extended as follows.

```
type t_readable, t_writable, t_sysbin, t_dte, t_generic, t_log;
domain d_daemon =  (/sbin/init),
                   (crwd->t_writable),
                   (rxd->t_readable),
                   (rd->t_generic, t_dte, t_sysbin),
                    (auto->d_login, d_log);
domain d_log =      (/usr/sbin/syslogd),
                    (crwd->t_log),
                    (rwd->t_writable),
                    (rd->t_generic, t_readable);
assign -r t_log /usr/var/log;
assign t_writable /usr/var/log/wtmp, /usr/var/log/utmp;
```

If a process in the domain **d_daemon** invokes the **syslogd** process, the **syslogd** process enters the **d_log** domain. It can now manipulate system logs and can read and write writable logs but cannot access system executables. If a user tries to manipulate a log object, the request is denied. The **d_user** domain gives its subjects no rights over **t_log** objects.

4.5.2. Low-Level Policy Languages

A low-level policy language is simply a set of inputs or arguments to commands that set, or check, constraints on a system.

EXAMPLE: The UNIX-based windowing system X11 provides a language for controlling access to the console (on which X11 displays its images). The language consists of a command, **xhosts**, and a syntax for instructing the command to allow access based on host name (IP address). For example,

```
xhost +groucho −chico
```

sets the system so that connections from the host **groucho** are allowed but connections from **chico** are not.

EXAMPLE: File system scanning programs check conformance of a file system with a stated policy. The policy consists of a database with desired settings. Each scanning program uses its own little language to describe the settings desired.

One such program, **tripwire** [569], assumes a policy of constancy. It records an initial state (the state of the system when the program is first run). On subsequent runs, it reports files whose settings have changed.

The policy language consists of two files. The first, the **tw.config** file, contains a description of the attributes to be checked. The second, the database, contains the values of the attributes from a previous execution. The database is kept in a readable format but is very difficult to edit (for example, times of modification are kept using base 64 digits). Hence, to enforce conformance with a specific policy, an auditor must ensure that the system is in the desired state initially and set up the **tw.config** file to ignore the attributes not relevant to the policy.

The attributes that **tripwire** can check are protection, file type, number of links, file size, file owner, file group, and times of creation, last access, and last modification. **Tripwire** also allows the cryptographic checksumming of the contents of the file. An example **tripwire** configuration file looks like

```
/usr/mab/tripwire-1.1 +gimnpsu012345678—a
```

This line states that all attributes are to be recorded, including all nine cryptographic checksums, but that the time of last access (the "a") is to be ignored (the "-"). This applies to the directory and to all files and subdirectories contained in it. After **tripwire** is executed, the database entry for that README file might be

```
/usr/mab/tripwire-1.1/README 0 ..../. 100600 45763 1 917 10
33242 .gtPvf .gtPvY .gtPvY 0 .ZD4cc0Wr8i21ZKaI..LUOr3
.0fwo5:hf4e4.8TAqd0V4ubv ?...... ...9b3
1M4GX01xbGIX0oVuGo1h15z3 ?:Y9jfa04rdzM1q:eqt1APgHk
?.Eb9yo.2zkEh1XKovX1:d0wF0kfAvC
?1M4GX01xbGIX2947jdyrior38h15z3 0
```

Clearly, administrators are not expected to edit the database to set attributes properly. Hence, if the administrator wishes to check conformance with a particular **policy** (as opposed to looking for changes), the administrator must ensure that the system files conform to that policy and that the configuration file reflects the attributes relevant to the policy.

EXAMPLE: The RIACS file system checker [105] was designed with different goals. It emphasized the ability to set policy and then check for conformance. It uses a database file and records fixed attributes (with one exception—the cryptographic checksum). The property relevant to this discussion is that the database entries are easy to understand and edit:

```
/etc/pac 0755 1 root root 16384 12 22341 Jan 12, 1987 at 12:47:54
```

The attribute values follow the file name and are permissions, number of links, owner and group, size (in bytes), checksum, and date and time of last modification. After generating such a file, the analyst can change the values as appropriate (and replace those that are irrelevant with a wild card "*"). On the next run, the file system state is compared with these values.