USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

Modular Construction of DTE Policies

Serge E. Hallyn
IBM Linux Technology Center
Austin, TX 78759
hallyn@cs.wm.edu

Phil Kearns
College of William and Mary
Williamsburg, VA 23185
kearns@cs.wm.edu

Abstract

This paper describes a tool which composes a policy for a fine-grained mandatory access control system (DTE) from a set of mostly independent policy modules. For a large system with many services, a DTE policy becomes unwieldy. However, many system services and security extensions can be considered to be largely standalone. By providing for explicit grouping, namespaces, and globbing by namespaces, inter-module access rules can be made generic enough to permit modules to be mixed and matched as needed. As a result, it becomes easier to extend a policy, debug a policy, and to distribute meaningful policy modules with new software.

1 Introduction

Domain and Type Enforcement (DTE) [1] is a fine-grained mandatory access control system. An implementation exists for Linux as a Loadable Security Module (LSM) [16]. The DTE LSM reads the policy it enforces through a text file through sysfs. The policy language closely resembles TIS's original DTEL policy language, which was explicitly intended to be intuitive to read and write. We have previously presented tools to analyze and edit policies. We now present a tool to compose policies from policy modules, which are smaller, simpler policy excerpts. In practice, we find policy modules far simpler to work with than a single large policy.

We begin by describing DTE and DTE policies in more detail. Next we describe the syntax of a policy module. We describe methods of grouping types and domains, the priority assigned to access rules based upon source and target, and hooks for system interaction during policy compilation. Then we describe a ftpd protection policy previously presented[7], and show how the ftp-relevant portion of this policy becomes a module.

2 DTE

DTE specifies two types of labels, called types and domains. It assigns types to files, and domains to processes. File access is controlled from domains to types, and signal access is controlled between processes in different domains. A process may transition to a new domain only on a call to `execve`. A domain may only be entered through files labeled with types explicitly marked as entry types for that domain. These files are called entry points for that domain. Domains may only transition to certain other domains. There are two types of domain transitions. The first, called `exec`, is voluntary, while the second, called `auto`, is mandatory. When a process under some domain executes a file which is an entry point to another domain, to which the first domain has auto access, then the process will transition to the new domain. If the file was an entry point to another domain to which the first had exec access, then the process ordinarily does not switch domains. It may request a domain transition by performing

```
echo -n <new_domain> > \
      /proc/<pid>/attr/exec
```

before executing the new file.

The DTE policy file specifies all types, all domains, the file system's type assignments, domain to type access, domain signal access, permitted domain transitions, and domain entry points. See [7] for more details about the policy file.

Policies for fine-grained MAC systems are mostly constructed as one unit and by hand. For instance, a massive effort is under way to create a complete, safe, SELinux policy for several distributions of Linux [13, 14]. Tools exist [8] for analyzing DTE policies, and such work is also being done for policies of other fine-grained MAC systems [9, 12, 15]. Nevertheless, working with a large policy remains a painful experience. However, when working with large policies, patterns begin to emerge. Policies typically consist of several sets of domains and types. The entities within a set work together to achieve some goal, and the sets often interact very little. For instance, in the ftp policy presented in [7], the domain `ftpd_d`, and the types `ftpd_t` and `ftpd_xt`, work together to protect the system from an unsafe binary. By removing these entities, and all references to them, the remaining policy becomes simpler. We call this collection of domains, types, and all access rules pertaining to them, a module. The ftp module is shown in Figure 3, and will be described in Section 4.

Allowing policies to be composed from simple, meaningful, and coherent pieces will serve several purposes.

First, creation of policies will become far more efficient. For instance, when adding a new domain to an existing policy, one might have to enter hundreds of type accesses in order to get it properly interacting with the current policy. In contrast, modules allow domains and types to be grouped at several levels, and access to be specified using any of these groups.

Second, adding a feature to a policy, such as a new method of controlling access to the shadow file, or protection from a critical binary in which an as-yet unsolved vulnerability has been found, will become a simpler task. The module can be written entirely from its own point of view. Furthermore, in researching the state of the current policy, in order to understand how to properly insert a new feature, one need only look at those modules which can affect the new functionality.

Third, modules may be helpful in simplifying the analysis, and proof of invariants, of policies. For instance, several modules may be trivially shown to be irrelevant to the ability of the `inetd` daemon, if remotely exploited, to erase the `utmp` log file.

Finally, because a module generally encodes domains, types, and access rules which work together toward some end, it is a natural way to express the security policy changes necessary for a new piece of software. Software companies and free software groups, therefore, could distribute policy modules along with software packages.

3 Module File Specification

We now discuss the structure of a module file. The module syntax specification follows.

```
<module_file> ::=
  <module>+

<module> ::=
  Module <mod_name>
    [<domain_def>|<type_def>| \
    <group_def>]+
  end

<domain_def> ::=
  domain <dom_name>
    <dom_line>+
  end
```

A module file may contain more than one module. Each module may contain several domain, type, and group definitions, as well as the access rules pertaining to them.

```
<dom_line> ::=
  entries <type_name>+  |
  [absolute] signal [in|out] <gen_dom> \
      <sig_num>  |
  [absolute] domain [in|out] <gen_dom> \
      [auto|exec|none] |
  [absolute] type <gen_type> <type_acc>|
  assert <policy_name> <data> |
  DEFAULT_DOMAIN
```

The domain definitions declare a unique name for the domain, a set of entry types, and a set of access rules pertaining to the new domain. Domain transition or signal access rules may be `in`, in which case they specify access from other domains to the new domain, or they may be `out`, defining access from the new domain to other domains. Since types are passive objects, which cannot themselves access other types or domains, the type access rules in a domain definition do not include the `in` or `out` keyword.

Exactly one domain definition applied to a policy must contain the keyword `DEFAULT_DOMAIN`. That domain will be assigned to the first process on the system.

```
<type_def> ::=
  type <type_name>
    <type_line>+
  end

<type_line> ::=
  <path_type> <path_name>+ |
  [absolute] access <gen_dom> <type_acc>|
  <default_type>  |
  assert <policy_name> <data>

<default_type> ::=
  DEFAULT_ETYPE | DEFAULT_UTYPE | \
      DEFAULT_RTYPE
```

Type definitions declare a unique name for the type, a set of paths assignment rules, and a set of access rules. Clearly, the access rules are only incoming from domains. A type whose definition contains the keyword `DEFAULT_RTYPE` will be assigned to the root of the file system, and recursively to its descendants until another type assignment rule applies. Alternatively, one type may be labeled as `DEFAULT_ETYPE`, and another may be labeled as `DEFAULT_UTYPE`. The first will be assigned to the root of the file system, and the second will be assigned to its descendants until another type assignment rule applies.

Both type and domain definitions may contain `assert` statements. These are used for maintenance of policy constraints. They are stored with the type definition until module application, but their interpretation and enforcement is defined by the named policy consistency class, any number of which may be written by the policy authors to ensure the maintenance of any module properties. The last line of the `ftpd_xt` type definition in Figure 3 is an example of an assert statement,

instructing a module loaded as `blp` to label this type as protected.

```
<group_def> ::=
  group domain <dom_name>
    import <dom_name>+
  end

<group_def> ::=
  group type <type_name>
    import <type_name>+
  end

<gen_dom>  ::= all | none | <dom_name>
<gen_type> ::= all | none | <type_name>
```

Grouping is accomplished on several levels. First, the keyword `all` refers to all domains or types which are currently known. Second, a group definition in a module may bind a name to a set of domains or types.

For instance, the following module segment defines a group of domains which may transition to user domains, and may require to files such as `.bashrc` and `.xsession`.

```
group login_domains_g
        import login_d su_d
end
```

A separate `X11` module might extend this group using

```
group login_domains_g extend
        import xdm_d
end
```

in order to borrow `login_d`'s and `su_d`'s rights to read user login files.

The following module segment defines a type which is actually called `root_t`.

```
type base.extraneous.root_t
        DEFAULT_RTYPE
        [...]
end
```

Since `root_t` is the type name which will be used in the final DTE policy, no names within the namespace may actually clash. Modules may refer to this type using any of the following names:

1. `all`
2. `base.+`
3. `base.extraneous.+`
4. `base.extraneous.*`
5. `base.extraneous.root_t`
6. `root_t`

In addition, any type groups which have imported this type can also be used to refer to this type.

The name `base.extraneous` may be a real type, or it may simply be a namespace placeholder, depending on whether any module defines a type by that name. A namespace placeholder is the parent of a domain, type, or group, which is not itself defined to be a domain, type, or group. It can be referred to during namespace globbing, but will not appear in the final policy.

Namespace globbing works as follows. When a name ends in `.+`, it refers to all descendants under this name. When a name ends in `.*`, it refers to only the immediate children of this name. Therefore `base.+` includes `base.extraneous.root_t`, but `base.*` does not. If `base.extraneous` were itself a type, then `base.*` would include this type, as would `base.+`.

Type of access	Priority level
Absolute single in	12
Absolute single out	11
Absolute group in	10
Absolute group out	9
Absolute all in	8
Absolute all out	7
Single destination in	6
Single destination out	5
Group in	4
Group out	3
Default (all) in	2
Default (all) out	1

Figure 1: Priorities of access rules

3.1 Priority of Access Rules

Since domains and types can declare conflicting access rules, we must clearly define the priority of access rules. Much thought has been given to the current priorities, which have been somewhat modified following experience with an earlier module compiler prototype. The priority takes the form of an integer between 1 and 12. The priority assigned to access rules is shown in Figure 1.

Each type of access consists of three pieces of information. First, it can be `in` or `out`. This is relative to the type or domain in which it is defined. When a domain specifies a certain type access, this is a `out` access rule, as the access is outbound from the domain. If a type defines access from some domain, this is `in`, as the access is inbound from the domain to the type. The second piece of information relates to the precision of the rule target. When an access rule names a specific domain or type, this is `single` access. If the rule names a group, or a namespace expansion such as `Services.*`, this is `group` access. If the rule targets the keyword `all`, this is of course `all` access. Finally, the rule is either `absolute` or not. This depends only upon whether the

access rule is preceded by the keyword `all`.

If two conflicting rules have been defined pertaining to the access permitted from a domain to another domain or type, then the rule with the highest priority will be applied. For instance, the base module's definition of type `base_t` specifies that all domains have `absolute` access `rxld` (read, execute, lookup, and descend) to `base_t`. This rule is `absolute all in`, and therefore has a priority of 8. Assume we write a new module, defining a domain intended to contain untrusted code. The domain definition might contain the statement:

```
absolute type all none
```

This rule is `absolute all out`, and therefore is priority 7. Since an `absolute all in` access rule has a higher priority than `absolute all out`, the new untrusted domain will receive `rxld` access to `base_t`, even though it asked for none. Had it in fact gotten none, then it would not be able to access any types at all, as it could not descend to them through the root of the file system. Similarly, if any types defined in the new module are intended to be accessed by the untrusted domain, then these types must specify incoming access from the untrusted domain as `absolute`, to ensure that that it will override the untrusted domain's outgoing type access definition. On the other hand, the base policy specifies a type `bin_t`, which includes a normal `group in` definition. As this is of a lower priority than `absolute out`, the access rule specified by the new module's untrusted domain is chosen, denying the untrusted domain all access to type `bin_t`. As we will see, this is a crucial element of the ftp module, preventing the ftp server from providing attackers with root shells, for instance.

Note that incoming access overrides outgoing access for the same target precision and `absolute` status. More specific rules override more general rules, unless the `absolute` keyword is present in one of the rules.

The usage of these keywords is intended to be intuitive. However, a switch to usage of simple numeric priority has not been ruled out. For instance, in place of

```
absolute domain in \
login_domains_grp auto
```

a module would specify

```
domain in login_domains_grp \
auto 60
```

.

The disadvantages to this are that module authors might require a deeper understanding of how policy compilation is affected by the priorities, and would need to consider these effects explicitly for each access rule.

3.2 Module Application

A set of modules may be applied simultaneously, and more than one set may be applied in series. For instance, we may begin by combining a set of base modules, then apply a set of service modules, and finally apply a module to ensure a particular security feature. We must therefore clearly define the behavior of group expansion across multiple module applications.

For named domain and type groups referenced in access rules, the group is expanded at the time of module application. In other words, for each member of the group, a new access rule is defined with the same access details as the original rule. Each newly created rule is associated with a `group` priority, to ensure proper resolution of any future conflicts. If the group has not yet been defined, an error is raised and compilation fails. For namespace globbing, that is, `*` and `+`, the currently defined descendants and children (respectively) of the parent being expanded are used. For instance, assume we applying a module which contains the rule

```
domain some_domain
      type base.exec.+ rwx
end
```

If the only children of `base.exec` defined thus far are the two types `base.exec.sbin` and `base.exec.bin`, then only these types are included in this rule. A later module may define type `base.exec.javabin`, but this type will not be added to the access rule.

The `all` target keyword is treated somewhat differently. An access rule directed at `all` will be expanded at the time of module application. Again the new access rules resulting from the expansion are stored with an `all` level for later conflict resolution. However, a generic form of the rule is also stored. All such generic rules are expanded each time a set of modules is applied. If the rule had not previously been applied, any policy consistency modules will be consulted at the new rule creation, just as with any other new access rule. For example, the base module defines default access `rld` to type `base_t` for `all` domains. This rule is expanded after each module application, so that all domains will be granted this access.

3.3 Keyword Substitution

One of the goals listed in Section 2 for the use of policy modules is to facilitate distribution of policy modules with new software. It must therefore be possible to apply policy modules across a variety of systems. To accomplish this in any meaningful way will often require some bit of system interaction. For instance, a policy module distributed with `xdm` might require labeling each user's `$HOME/.xsession` as an entry type

to the user domain. This requires system interaction to determine valid users on this system who actually have a `$HOME/.xsession` file.

The prototype module compiler provides system interaction through an `exec` keyword. This is augmented with looping support over variables which have been set using `exec`. Using these features, an excerpt of the xdm policy might look as follows:

```
1  define xsession_f exec /bin/ls \
2    /home/*/.xsession
3
4  type xdm_out_fromuser_et
5    epath /etc/X11/xdm/Xsession
6
7    foreach file ‘xsession_f‘
8      epath ‘file‘
9    endforeach file
10
11   access user_d rwxlcd
12   access login_domains_grp r
13 end
14
15 domain user_d extend
16   entries xdm_out_fromuser_et
17 end
```

The first command, on lines 1 and 2, assigns to the variable `xsession_f` the result of executing the command `/bin/ls /home/*/.xsession`. This will contain a list of all user `.xsession` files, one per line. Lines 7 through 9 loop over each line returned by the `ls` command, each time adding a new epath line to the `xdm_out_fromuser_et` type definition, and replacing `‘file‘` with the next file. The result is a type to which the user domain may write, and which those domains which are members of the `login_domains_grp` group may read. The last three lines extend the `user_d` domain such that other domains may transition into it by executing the `.xesssion` files which were found. Of course, in many cases more complicated calculations than a directory listing will be required. The output from any script or program can be assigned to variables. However, the use of complicated external scripts might add an unwelcome element of unpredictability to the policy creation process. The policy consistency classes will offer some support to system administrators trying to keep this in check, and graphical analysis tools will remain available for analyzing the final policy.

3.4 Inheritance

An issue which may deserve further consideration is that of inheritance. It would seem to make sense to construct the type namespace such that certain properties, perhaps `absolute` access rules, are automatically inherited by the children of a type. On the other hand, this may simply needlessly complicate the process of policy creation, the simplification of which is the precise goal of the policy compiler. Currently, the notion of inheritance does not exist in the module compiler.

4 Ftpd Protection Module

Ftp daemons provide a great deal of interaction, usually with completely unauthenticated, or anonymous, users. In order to permit user logins, however, some ftp daemons run as root. A programming error such as buffer overflow or string format vulnerability can therefore lead to the execution of arbitrary commands using superuser privileges by anyone on the internet.

4.1 Original Policy

Figure 2 demonstrates policy to protect a system from ftpd. While a DTE system could actually boot and run with this policy, it is a minimalist policy designed only to protect from ftpd. An actual useful policy would be much larger, but contain a nearly identical set of ftp protections. The policy provides protection from attackers by containing the ftp daemon to a domain, called `ftpd_d`, which has limited access rights. This domain is not allowed to transition into any other domains, so that any code executed (legitimately or not) by the ftp daemon will also be subject to the same access restrictions. The domain is automatically entered whenever a privileged process executes `/usr/sbin/in.ftpd`. It requires permission to execute its entry point, library files, and files located under `/home/ftp/bin`. It needs read and write access to devices, `/home/ftp/incoming`, a transfer log, and some temporary files. The domain is refused the ability to execute anything it might have written. It has permission to read under `/home/ftp`, `/etc`, and, unfortunately, the password and shadow files. However, it lacks permissions to execute files under `/bin`, `/usr/bin`, etc. Therefore all existing exploits, which require the ability to execute `"/bin/cat /etc/passwd"` or `/bin/sh`, will fail.

4.2 Ftp Module

We now separate the ftp functionality out from the policy and into a module. The ftp module is found in Figure 3. It again defines a `ftpd_d` domain, and `ftpd_t`, `ftpd_et`, `ftpd_xt`, and `ftpd_wt` types. The `ftpd_d` definition specifies inbound domain transitions from `boot_t`, and from all domains defined under `Admin.services`. `Ftpd_d` may not transition to any other domains, so this access rule is `absolute`. This does not completely rule out `ftpd_d` being permitted to transition to another domain. However, in order for `ftpd_d` to be allowed to transition to another

```
# ftpd protection policy
types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
      config_t ftpd_t ftpd_xt w_t
domains root_d login_d user_d ftpd_d
default_d root_d
default_et root_t
default_ut root_t
default_rt root_t
spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t \
      rwxcd->spool_t rwcdx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t \
      rwxcd->passwd_t rxwcd->shadow_t rwxcd->dev_t rwxcd->config_t \
      rwxcd->w_t) (auto->login_d auto->ftpd_d) (0->0)
spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd->spool_t \
      rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t rwxcd->dev_t \
      rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17->0)
spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rxwcd->shadow_t \
      rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rwxd->root_t \
      rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17->0)
spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwcd->ftpd_t rd->user_t rd->root_t \
      rxd->lib_t r->passwd_t r->shadow_t rwcd->dev_t rdx->ftpd_xt \
      rd->config_t rwcd->w_t d->spool_t) () (14->root_d 17->root_d)
assign -u /home user_t
assign -u /tmp spool_t
assign -u /var spool_t
assign -u /dev dev_t
assign -u /scratch user_t
assign -r /usr/src/linux user_t
assign -u /usr/sbin binary_t
assign -e /usr/sbin/in.ftpd ftpd_xt
assign -r /home/ftp/bin ftpd_xt
assign -e /var/run/ftp.pids-all ftpd_t
assign -r /home/ftp ftpd_t
assign -e /var/log/xferlog ftpd_t
assign -r /lib lib_t
assign -e /etc/passwd passwd_t
assign -e /etc/shadow shadow_t
assign -e /var/log/wtmp w_t
assign -e /var/run/utmp w_t
assign -u /etc config_t
```

Figure 2: A DTE policy to protect from *wu-ftpd*.

```
Module Service.ftp
     domain ftpd_d
          entries ftpd_et
          absolute domain out all none
          domain in boot_d auto
          domain in Admin.services.+ exec
          absolute type all none
          signal out boot_d 14,17
          signal out Admin.services.+ 14,17
     end

     type ftpd_t
          access all none
          absolute access ftpd_d rld
          rpath /home/ftp
     end

     type ftpd_et
          access all r
          absolute access ftpd_d rx
          epath /usr/sbin/in.ftpd
     end

     type ftpd_xt
          access all none
          absolute access ftpd_d rxld
          access root_d rwcld
          rpath /home/ftp/bin
          assert mblp protect
     end

     type ftpd_wt
          access all none
          absolute access ftpd_d rwcld
          rpath /home/ftp/incoming
     end
End
```

Figure 3: FTP Policy Module

domain, the other domain would have to explicitly ask for `ftpd_d` to be permitted to transition to it, or add `ftpd_d` to a group and provide that group with inbound transition access.

Type `ftpd_t` is located under `/home/ftp`. Only `ftpd_d` may observe this type, no one may modify or execute. The file `/usr/sbin/in.ftpd` is the entry type through which `ftpd_d` may be entered, signified both by the `ftpd_et` type definition, and the `entries` line in the `ftpd_d` definition. The files under `/home/ftp/bin`, labeled as `ftpd_xt`, may be executed by `ftpd_d`, and written by `root_d`. There is no single domain which may both modify and execute these files. Finally, the files located under `/home/ftp/incoming`, labeled `ftpd_wt`, may be written, but not executed by `ftpd_d`. It may not be accessed by any other domains.

This set of accesses was also accomplished using the ftp policy. In fact, the module will eventually be compiled into a policy. However, using the module, we are able to limit statements concerning `ftpd_wt` to the 6 simple lines which define the type, and trust that any domains which are later added under `Admin.services` will be able to transition to `ftpd_d`.

A detailed discussion of

```
assert mblp protect
```

is beyond the scope of this paper. However a brief explanation is appropriate. If no policy constraint class named `mblp` has been loaded, then this line will be ignored. If this class has in fact been loaded, then it is instructed to label this type, `ftpd_xt`, using the keyword `protect`. A class may do with this information what it likes. It will be called once before and once after each application of a set of modules, and given a copy of the policy at each point. The `mblp` class, in particular, will print a warning if any domain is in fact allowed to modify the protected type. This demonstrates the simplest use of policy consistency classes. We could in fact write a class to simply read assertions which must hold true in the policy. More interesting classes, such as `mblp`, compare calculations on the policy before and after module application.

The most significant advantage of separating the ftp module out from the base policy becomes apparent when we consider writing more modules. For instance, the base policy module does not allow users to change their passwords. To add this functionality, we use a module such as that in Figure 4. Nothing in the ftp module needs to change, and we do not need to consult the ftp module while writing the password module. In contrast, adding password functionality to an existing policy could become very invasive.

5 Control

The simplest way to compile DTE modules into a policy is to use the command line utility `dte_pc.py`. A list of the modules to be applied is placed into a file, which is given as a command line argument to `dte_pc.py`. The resulting policy is placed into a file also specified as an argument.

Using `dte_pc.py`, all modules are applied simultaneously. Greater control over module application can be had on the python command line, or by writing custom module application scripts. The following python lines, for instance, combine the two modules `base` and `user`, and then apply a third module, `ftp`.

```
from DTEModule import ModuleFile
# firstmods will be an array containing
# the "base" and "ftp" modules
firstmods = ModuleFile("base").Modules()
firstmods.extend(
       ModuleFile("user").Modules()
)
ftpmod = ModuleFile("ftp").Modules()
p = DTEPolicy.Policy()
# Apply "base" and "user" together
p.apply_modules(firstmods)
# Now apply "ftp" separately
p.apply_modules(ftpmod)
p.write("dte_output_file.conf")
```

One advantage of using this code is the enhanced precision in group definitions as described in Section 3.2. That is, if any groups are defined and referenced in `base` or `user`, and then extended in `ftp`, then the references to them in `base` and `user` will not include the members added in `ftp`. Additionally, policy consistency classes are only invoked before and after each DTEPolicy `apply_modules()` invocation, so the above code would force the application of `ftp` to be more closely scrutinized. If all modules are applied at once, then a policy consistency class will only ever compare an empty policy to the final policy, which may not be useful, depending upon the policy consistency class.

6 Related Work

The policy language read by the DTE module is based in large part on the DTEL policy language used by the original DTE on Unix implementation [1]. The policy consistency classes and related assert statements are a generalization of the ideas proposed in [2]. Here Bell-LaPadula [4] and Strict Integrity [3] relations, assured pipelines [5], and the Clark-Wilson [6] concepts of constrained data items (CDIs) and transformation procedures (TPs) were used to guarantee maintenance of certain properties through dynamic policy changes. OO-DTE [11] applied DTE to CORBA distributed objects, and introduced an object oriented policy language,

```
Module password
     # domains passw_d
     # types passw_t passw_et shadow_t
     type passw_et
          epath /bin/passw
          access all rx
          access Admin.admins rwxlcd
     end

     type passw_t
          epath /etc/passwd /etc/passwd.tmp /etc/.pwd.lock
          access all r
          access passw_d rw
     end

     type shadow_t
          epath /etc/shadow
          access all none
          access login_domains_grp r
          access passw_d rw
     end

     domain passw_d
          type conf_t rlcd
          entries passw_et

          domain in all auto
          domain out all none
     end

End
```

Figure 4: Password Policy Module

DTEL++. In OO-DTE, a user's domain was used to determine permission to execute or implement methods assigned to particular types. This is quite different from the meaning of DTE in a operating system such as Linux.

SELinux policies [13], like DTE policy modules, are compiled, in this case to a binary policy file. The SELinux policy makes liberal use of macros, which are defined throughout the policy, and compiled using the `m4` preprocessor. SELinux policies are less structured than policy modules. There is no sense of domains and types being objects, of access rules belonging to the definition of the source of target of the definition, or of priority of conflicting access rules. SELinux policies make use of simple `assert` rules for safety constraints, but no attempts have been made to provide more in-depth analysis of the effects of a particular piece of policy during compilation. SELinux policies are more detailed and more complicated than DTE policies. The possibility and usefulness of transcribing the idea of policy modules to SELinux policies while keeping modules readable and small, remains to be investigated.

Tools exist to aid in editing and analyzing DTE and SELinux policies [8, 12, 15]. These tools analyze whole policies, and therefore complement, rather than compete with the DTE policy modules concept. The policy consistency classes used by the module compiler are designed to analyze the effect of particular policy enhancements on the overall policy. The existing DTE policy analysis tools can still be used on the policies resulting from module compilation.

IBM Research is investigating the concept of access control spaces [10], and working toward a method to determine whether an SELinux policy satisfies certain integrity goals [9]. This work again analyzes whole policies. Ultimately, it is possible that Linux vendors could use this approach to verify the correctness of a TCB included with their distribution, while system administrators could use policy consistency classes to analyze the effects of their own policy modules on the base policy.

7 Conclusion

By the very virtue of being fine-grained, policies for MAC systems such as DTE and SELinux become very large, currently tens of thousands of lines for SELinux. Policy modules break this into a number of smaller pieces, and permit authors to intelligently group objects and subjects to permit concise and expressive access rules. The careful construction of a policy module language results in a far more convenient, more efficient, and safer policy specification. In practice, it has greatly eased the movement by the authors between various testing and development machines with various distributions.

8 Availability

The DTE LSM is available as part of the LSM project at `http://lsm.immunix.org`. The policy compiler is available from `http://www.nekonoken.org`. Both are licensed under the GPL.

9 Acknowledgments

Hallyn's work was supported in part by a USENIX Scholarship. The authors also wish to thank the paper shepherd, Crispin Cowan, for his helpful and constructive comments.

References

[1] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat, *A Domain and Type Enforcement UNIX Prototype*, Usenix Security Symposium (1995).

[2] Tim Fraser and Lee Badger, *Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE*, Proceedings of IEEE Symposium on Research in Security and Privacy, 1998.

[3] K. J. Biba, *Integrity Considerations for Secure Computer Systems*, Mitre Technical Report ESD-TR-76-372, 1977.

[4] D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, Mitre Technical Report ESD-TR-75-306, 1976.

[5] W.E. Boebert and R.Y. Kain, *A Practical Alternative to Hierarchical Integrity Policies*, Proceedings of the National Computer Security Conference, 1985.

[6] David D. Clark and David R. Wilson, *A Comparison of Commercial and Military Computer Security Policies*, Proceedings of the IEEE Symposium on Security and Privacy, 1987.

[7] Serge Hallyn and Phil Kearns, *Domain and Type Enforcement for Linux*, ALS 2000.

[8] Serge Hallyn and Phil Kearns, *Tools to Administer Domain and Type Enforcement*, LISA 2001, p. 151-156.

[9] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang, *Analyzing Integrity Protection in the SELinux Example Policy*, Usenix Security Symposium, 2003.

[10] Trent Jaeger and Xiaolan Zhang, *Policy Management using Access Control Spaces*, ACM Transactions on Information and System Security, 2003.

[11] Durward McDonnel and David Sames and Gregg Tally and Robb Lyda, *Security for Distributed Object-Oriented Systems*, DARPA Information Survivability Conference and Exposition, June 2001.

[12] MITRE Corporation, *SLAT: Information Flow Analysis in Security Enhanced Linux*, available at `http://www.nsa.gov/selinux`.

[13] Stephen Smalley, *Configuring the SELinux Policy*, NSA Technical Report, `http://www.nsa.gov/selinux/papers/policy2-abs.cfm`.

[14] Stephen Smalley et al, *SELinux mailing list*, `http://www.nsa.gov/selinux/list-archive/summary.cfm`.

[15] Tresys Technology, *Security-enhanced Linux Policy Tools*, `http://www.tresys.com/selinux/`, 2003.

[16] Christ Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman, *Linux Security Modules: General Security Support for the Linux Kernel*, Usenix Security Symposium, 2002.