Contents

Introduction **5**

Design Philosophy 6

Goals and Primary Features 7

Design Overview 8

CIL Information **9**

Declarations 9

Definitions 10

Symbol Character Set 11

String Character Set 11

Comments 11

Namespaces 11

Global Namespace 12

Expressions 13

Name String 15

Access Vector Rules **16**

allow 17

auditallow 18

dontaudit 18

neverallow 19

deny 19

allowx 20

auditallowx 21

dontauditx 21

neverallowx 22

Call / Macro Statements **22**

call 22

macro 23

Class and Permission Statements **25**

common 25

classcommon 26

class 27

classorder 28

classpermission 29

classpermissionset 29

classmap 31

classmapping 32

permissionx 33

Conditional Statements **35**

boolean 35

booleanif 35

tunable 37

tunableif 37

Constraint Statements **39**

constrain 39

validatetrans 41

mlsconstrain 42

mlsvalidatetrans 44

Container Statements **46**

block 46

blockabstract 47

blockinherit 48

optional 50

in 51

Context Statement **52**

context 52

Default Object Statements **53**

defaultuser 53

defaultrole 54

defaulttype 55

defaultrange 56

File Labeling Statements **57**

filecon 57

fsuse 58

genfscon 60

Multi-Level Security Labeling Statements **61**

sensitivity 61

sensitivityalias 62

sensitivityaliasactual 62

sensitivityorder 63

category 63

categoryalias 64

categoryaliasactual 64

categoryorder 65

categoryset 66

sensitivitycategory 67

level 68

levelrange 69

rangetransition 71

Network Labeling Statements **72**

ipaddr 72

netifcon 73

nodecon 74

portcon 75

Policy Configuration Statements **76**

mls 76

handleunknown 77

policycap 77

Role Statements **78**

role 78

roletype 79

roleattribute 79

roleattributeset 80

roleallow 81

roletransition 82

rolebounds 83

SID Statements **83**

sid 83

sidorder 84

sidcontext 85

Type Statements **85**

type 85

typealias 86

typealiasactual 86

typeattribute 87

typeattributeset 88

expandtypeattribute 89

typebounds 90

typechange 91

typemember 92

typetransition 93

typepermissive 95

typeneveraudit 95

User Statements **96**

user 96

userrole 96

userattribute 97

userattributeset 98

userlevel 99

userrange 99

userbounds 100

userprefix 101

selinuxuser 102

selinuxuserdefault 103

Infiniband Statements **103**

ibpkeycon 104

ibendportcon 104

Xen Statements **105**

iomemcon 105

ioportcon 106

pcidevicecon 106

pirqcon 107

devicetreecon 108

Example Policy **108**

Introduction

The SELinux Common Intermediate Language (CIL) is designed to be a language that sits between one or more high level policy languages (such as the current module language) and the low-level kernel policy representation. The intermediate language provides several benefits:

- Enables the creation of multiple high-level languages that can both consume and produce language constructs with more features than the raw kernel policy (e.g., interfaces). Pushing these features into CIL enables cross-language interaction.

- Eases the creation of high-level languages, encouraging the creation of more domain specific policy languages (e.g., CDS Framework, Lobster, and Shrimp).
- Provides a semantically rich representation suitable for policy analysis, allowing the analysis of the output of multiple high-level languages using a single analysis tool set without losing needed high-level information.

Design Philosophy

CIL is guided by several key decision principles:

- Be an intermediate language - provide rich semantics needed for cross-language interaction but not for convenience. If a feature can be handled by a high-level language without sacrificing cross-language interoperability leave the feature out. Less is more.
- Facilitate easy parsing and generation - provide clear, simple syntax that is easy to parse and to generate by high-level compilers, analysis tools, and policy generation tools. Machine processing should be prioritized higher than human processing when there is a conflict as humans should be reading and writing high-level languages instead.
- Fully and faithfully represent the kernel language - the ultimate goal of CIL is the generation of the policy that will be enforced by the kernel. That policy must be full represented so that all of the policy can be represented in CIL. And that representation should not adorn, obscure, or otherwise hide the kernel policy. CIL should allow additional high-level language semantics but should not abstract away the essence of the kernel enforcement. Be C (portable assembler) not a pure functional language (which hides how the processor actually works).
- The only good binary file format is a non-existent one - CIL is meant for a source policy oriented world, so assume and leverage that. The only binary policy format moving forward should be for communication with the kernel.
- Enable backwards compatibility but don't be a slave to it - source, but not binary, compatibility with existing policies is a goal but not an absolute requirement. Where necessary it is assumed that manual or automated policy conversion will be required to move to enable the freedom needed to make CIL compelling.
- Don't fix what isn't broken - CIL is an opportunity to make bold changes to SELinux policy, but there is no reason to re-think core concepts that are working well. All changes to existing language constructs need a clear and compelling reason. One key aspect of the current policy to retain is it's order-independent, declarative style.

- No more M4 - the pervasive use of M4 and pre-processing in general has eased policy creation, but the side-effects cause many additional problems. CIL should eliminate the need for a pre-processor.
- Shift more compilation work to happen per-module instead of globally - the current toolchain performance is often driven by the size of the policy and the need to have the entire policy loaded to do much of the processing. If possible, make it possible to do more compilation of one module at a time to increase performance. At the very least, clearly identify and manage language constructs that cause work on the global policy.

Goals and Primary Features

CIL is meant to enable several features that are currently difficult or impossible to achieve with the current policy languages and tools. While generality is always a goal, with CIL there are also several well-known and clear motivating language needs.

- Policy customization without breaking updates - one of the challenges in SELinux is allowing a system builder or administrator to change the access allowed on a system - including removing unwanted access - while not preventing the application of future policy updates from the vendor. It is desirable, therefore, to allow an administrator to make changes to vendor policy without necessitating the direct modification of the shipped policy files. This is most clearly seen when an administrator wants to remove access allowed by a vendor policy that is not already controlled by a policy boolean.
- Interfaces as a first class feature - interfaces, and macros before them, have been a successful mechanism to allow policy authors to define related sets of access and easily grant that access to new types. However, this success has been hampered by interfaces existing solely as pre-processor constructs, preventing compilers, management tools, and analysis tools from understanding them. This has many unintended consequences, including the need to recompile all modules to include the changes to an interface. Interfaces or some similar construct should become first class language features.
- Rich policy relationships - templates, interfaces, and attributes are currently the only means of quickly creating new types or sets of types with commonly needed access. However, use of these constructs require upfront design by the policy developer, limiting their use by system builders and administrators to rapidly create or mold existing policy. Policy authors need language features to create new types or modules based upon existing ones with large or small changes. These features should allow ad-hoc creation of new policy modules or types related to existing types.

- Support for policy management - semanage and related tools currently make policy modifications using private data stores and code to directly manipulate the binary policy format before it is generated for loading into the kernel. These tools should be able to generate and consume CIL to accomplish the same goals.

Design Overview

The design is aims to provide simplicity in several ways:

1. The syntax is extremely regular and easy to parse being based upon s-expressions.
2. The statements are reduced to the bare minimum. There is one - and only one - way to express any given syntax.
3. The statements are unambiguous and overlap in very well defined ways. This is in contrast to the current language where a statement, such as a role statement, might be a declaration, a further definition, or both depending on context.

The language, like the existing policy languages, is declarative. It removes all of the ordering constraints from the previous languages. Finally, the language is meant to be processed in source form as a single compilation unit - there is no module-by-module compilation. This has advantages (no need for compiled disk representation, better error reporting, simpler processing) with the primary disadvantage of space. However, this is not a problem in practice as the linking process for the binary policy modules required the entire representation in memory as well. It is, in many ways, a natural result of the declarative nature of the language.

In many ways, this design document describes what is different between the current language and CIL. For example, types have exactly the same semantics as they currently do, CIL simply uses a different syntax for declaring and referencing them. Consequently, no space is spent describing the semantics of types and only a small amount of space spent discussing the new syntax separate from interaction with new CIL features. Contrastingly, CIL has new constructs for creating, managing, and traversing namespace. There is a corresponding amount of space describing the semantics of those features.

When referring to current semantics it is important to note that there are currently three separate policy languages in common usage: the reference policy syntax created in M4 (which includes interfaces and templates), the module syntax understood by checkmodule, and what is commonly called the kernel policy which is the policy understood by checkpolicy. In general, CIL preserves the current kernel policy almost unchanged (just with different syntax) and layers on features from the module language, reference policy, and novel new features.

When discussing current semantics, if the context is not clear attempts will be made to clarify which policy language is being referenced.

CIL Information

1. Not all possible alternate statement permutations are shown, however there should be enough variation to work out any other valid formats. There is also an example `policy.cil` file in the test directory.
2. The MLS components on contexts and user statements must be declared even if the policy does not support MCS/MLS.
3. The CIL compiler will not build a policy unless it also has as a minimum: one `allow` rule, one `sid`, `sidorder` and `sidcontext` statement.
4. The role `object_r` must be explicitly associated to contexts used for labeling objects. The original `checkpolicy(8)` and `checkmodule(8)` compilers did this by default - CIL does not.
5. Be aware that CIL allows `class` statements to be declared in a namespace, however the policy author needs to note that applications (and the kernel) generally reference a class by its well known class identifier (e.g. `zygote`) however if declared in a namespace (e.g. `(block zygote (class zygote (...)))` or `(block zygote (class class (...)))`) it would be prefixed with that namespace (e.g. `zygote.zygote` or `zygote.class`). Unless the application / kernel code was updated the class would never be resolved, therefore it is recommended that classes are declared in the global namespace.
6. Where possible use `typeattribute`'s when defining source/target `allow` rules instead of multiple `allow` rules with individual `type`'s. This will lead to the generation of much smaller kernel policy files.
7. The site explains the language however some of the statement definitions are dated.

Declarations

Declarations may be named or anonymous and have three different forms:

1. Named declarations - These create new objects that introduce a name or identifier, for example:

   `(type process)` - creates a `type` with an identifier of `process`.

   `(typeattribute domain)` - creates a `typeattribute` with an identifier of `domain`.

`(class file (read write))` - creates a `class` with an identifier of `file` that has `read` and `write` permissions associated to it.

The list of declaration type statement keywords are:

block optional common class classmap classmapping sid user role roleattribute type classpermission classpermissionset typeattribute typealias tunable sensitivity sensitivityalias category categoryalias categoryset level levelrange context ipaddr macro policycap

2. Explicit anonymous declarations - These are currently restricted to IP addresses where they can be declared directly in statements by enclosing them within parentheses e.g. `(127.0.0.1)` or `(::1)`. See the Network Labeling Statements section for examples.

3. Anonymous declarations - These have been previously declared and the object already exists, therefore they may be referenced by their name or identifier within statements. For example the following declare all the components required to specify a context:

```
(sensitivity s0)
(category c0)
(role object_r)

(block unconfined
    (user user)
    (type object)
)
```

now a `portcon` statement can be defined that uses these individual components to build a context as follows:

```
(portcon udp 12345 (unconfined.user object_r unconfined.object ((s0) (s0(c0)))))
```

Definitions

Statements that build on the objects, for example:

- `(typeattributeset domain (process))` - Adds the `type` '`process`' to the `typeattribute` '`domain`'.
- `(allow domain process (file (read write))))` - Adds an `allow` rule referencing `domain`, `process` and the `file class`.

Definitions may be repeated many times throughout the policy. Duplicates will resolve to a single definition during compilation.

Symbol Character Set

Symbols (any string not enclosed in double quotes) must only contain alphanumeric `[a-z A-Z] [0-9]` characters plus the following special characters: `[].@=/*-_$%+-!|&^:~`#{}'<>?,`

However symbols are checked for any specific character set limitations, for example:

- Names or identifiers must start with an alpa character `[a-z A-Z]`, the remainder may be alphanumeric `[a-z A-Z] [0-9]` characters plus underscore `[_]` or hyphen `[-]`.
- IP addresses must conform to IPv4 or IPv6 format.
- Memory, ports, irqs must be numeric `[0-9]`.

String Character Set

Strings are enclosed within double quotes (e.g. `"This is a string"`), and may contain any character except the double quote (“).

Comments

Comments start with a semicolon ‘`;`’ and end when a new line is started.

Namespaces

CIL supports namespaces via containers such as the `block` statement. When a block is resolved to form the parent / child relationship a dot ‘`.`’ is used, for example the following `allow` rule:

```
(block example_ns
    (type process)
    (type object)
    (class file (open read write getattr))

    (allow process object (file (open read getattr)))
)
```

will resolve to the following kernel policy language statement:

```
allow example_ns.process example_ns.object : example_ns.file { open read getattr };
```

Global Namespace

CIL has a global namespace that is always present. Any symbol that is declared outside a container is in the global namespace. To reference a symbol in global namespace, the symbol should be prefixed with a dot '.' as shown in the following example:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This example has three namespace 'tmpfs' types declared:
;    1) Global .tmpfs
;    2) file.tmpfs
;    3) other_ns.tmpfs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; This type is the global tmpfs:
(type tmpfs)

(block file
    ; file namespace tmpfs
    (type tmpfs)
    (class file (open read write getattr))

    ; This rule will reference the local namespace for src and tgt:
    (allow tmpfs tmpfs (file (open)))
    ; Resulting policy rule:
    ; allow file.tmpfs file.tmpfs : file.file open;

    ; This rule will reference the local namespace for src and global for tgt:
    (allow tmpfs .tmpfs (file (read)))
    ; Resulting policy rule:
    ; allow file.tmpfs tmpfs : file.file read;

    ; This rule will reference the global namespace for src and tgt:
    (allow .tmpfs .tmpfs (file (write)))
    ; Resulting policy rule:
    ; allow tmpfs tmpfs : file.file write;

    ; This rule will reference the other_ns namespace for src and
    ; local namespace for tgt:
    (allow other_ns.tmpfs tmpfs (file (getattr)))
    ; Resulting policy rule:
    ; allow other_ns.tmpfs file.tmpfs : file.file getattr;
)

(block other_ns
```

```
    (type tmpfs)
)
```

Should the symbol not be prefixed with a dot, the current namespace would be searched first and then the global namespace (provided there is not a symbol of that name in the current namespace).

Expressions

Expressions may occur in the following CIL statements: `booleanif`, `tunableif`, `classpermissionset`, `typeattributeset`, `roleattributeset`, `categoryset`, `constrain`, `mlsconstrain`, `validatetrans`, `mlsvalidatetrans`

CIL expressions use the prefix or Polish notation and may be nested (note that the kernel policy language uses infix notation). The syntax is as follows, where the parenthesis are part of the syntax:

```
expr_set = (name ... | expr ...)
expr = (expr_key expr_set ...)
expr_key = and | or | xor | not | all | eq | neq | dom | domby | incomp | range
```

The number of `expr_set`'s in an `expr` is dependent on the statement type (there are four different classes as defined below) that also influence the valid `expr_key` entries (e.g. `dom`, `domby`, `incomp` are only allowed in constraint statements).

expr_key	classpermissionset roleattribute-set typeat-tributeset	categoryset	booleanif tunableif	constrain mlsconstrain validatetrans mlsvalidate-trans
dom				**X**
domby				**X**
incomp				**X**
eq			**X**	**X**
ne			**X**	**X**
and	**X**	**X**	**X**	**X**
or	**X**	**X**	**X**	**X**
not	**X**	**X**	**X**	**X**
xor	**X**	**X**	**X**	
all	**X**	**X**		
range		**X**		

1. The `classpermissionset`, `roleattributeset` and `typeattributeset` statements allow `expr_set` to mix names and `expr`s with `expr_key` values of: `and`, `or`, `xor`, `not`, `all` as shown in the examples:

This example includes all `fs_type type` entries except `file.usermodehelper` and `file.proc_security` in the associated `typeattribute` identifier `all_fs_type_except_usermodehelper_and_proc_security`:

```
(typeattribute all_fs_type_except_usermodehelper_and_proc_security)

(typeattributeset all_fs_type_except_usermodehelper_and_proc_security
    (and
        (and
            fs_type
            (not file.usermodehelper)
        )
        (not file.proc_security)
    )
)
```

The `cps_1 classpermissionset` identifier includes all permissions except `load_policy` and `setenforce`:

```
(class security (compute_av compute_create compute_member check_context load_policy

(classpermission cps_1)

(classpermissionset cps_1 (security (not (load_policy setenforce))))
```

This example includes all permissions in the associated `classpermissionset` identifier `security_all_perms`:

```
(class security (compute_av compute_create compute_member check_context load_policy
    compute_relabel compute_user setenforce setbool setsecparam setcheckreqprot
    read_policy)
)

(classpermission security_all_perms)

(classpermissionset security_all_perms (security (all)))
```

2. The `categoryset` statement allows `expr_set` to mix names and `expr_key` values of: `and`, `or`, `not`, `xor`, `all`, `range` as shown in the examples.

   Category expressions are also allowed in `sensitivitycategory`, `level`, and `levelrange` statements.

3. The `booleanif` and `tunableif` statements only allow an `expr_set` to have one `name` or `expr` with `expr_key` values of `and`, `or`, `xor`, `not`, `eq`, `neq` as shown in the examples:

```
(booleanif disableAudio
    (false
        (allow process device.audio_device (chr_file_set (rw_file_perms)))
    )
)

(booleanif (and (not disableAudio) (not disableAudioCapture))
    (true
        (allow process device.audio_capture_device (chr_file_set (rw_file_perms)))
    )
)
```

4. The `constrain`, `mlsconstrain`, `validatetrans` and `mlsvalidatetrans` statements only allow an `expr_set` to have one `name` or `expr` with `expr_key` values of `and`, `or`, `not`, `all`, `eq`, `neq`, `dom`, `domby`, `incomp`. When `expr_key` is `dom`, `domby` or `incomp`, it must be followed by a string (e.g. `h1`, `l2`) and another string or a set of `name`s. The following examples show CIL constraint statements and their policy language equivalents:

```
; Process transition:  Require equivalence unless the subject is trusted.
(mlsconstrain (process (transition dyntransition))
    (or (and (eq h1 h2) (eq l1 l2)) (eq t1 mlstrustedsubject)))

; The equivalent policy language mlsconstrain statement is:
;mlsconstrain process { transition dyntransition }
;    ((h1 eq h2 and l1 eq l2) or t1 == mlstrustedsubject);

; Process read operations: No read up unless trusted.
(mlsconstrain (process (getsched getsession getpgid getcap getattr ptrace share))
    (or (dom l1 l2) (eq t1 mlstrustedsubject)))

; The equivalent policy language mlsconstrain statement is:
;mlsconstrain process { getsched getsession getpgid getcap getattr ptrace share }
;    (l1 dom l2 or t1 == mlstrustedsubject);
```

Name String

Used to define `macro` statement parameter string types:

```
(call macro1("__kmsg__"))

(macro macro1 ((string ARG1))
    (typetransition audit.process device.device chr_file ARG1 device.klog_device)
)
```

Alternatively:

```
(call macro1("__kmsg__"))

(macro macro1 ((name ARG1))
    (typetransition audit.process device.device chr_file ARG1 device.klog_device)
)
```

Access Vector Rules

Rules involving a source type, a target type, and class permissions or extended permissions.

Rule definition:

```
(av_flavor source_id target_id|self|notself|other classpermission_id|permissionx_id)
```

Where:

av_flavor

The flavor of access vector rule. Possible flavors are allow, auditallow, dontaudit, neverallow, deny, allowx, auditallowx, dontauditx, and neverallowx.

source_id

A single previously defined source type, typealias or typeattribute identifier.

target_id

A single previously defined target type, typealias or typeattribute identifier.

Instead it can be one of the special keywords self, notself or other.

The self keyword may be used to signify that source and target are the same. If the source is an attribute, each type of the source will be paired with itself as the target. The notself keyword may be used to signify that the target is all types except for the types of the source. The other keyword may be used as a short-hand way of writing a rule for each type of the source where it is paired with all of the other types of the source as the target.

classpermission_id

A single named or anonymous classpermissionset or a single set of classmap/classmapping identifiers. Used for allow, auditallow, dontaudit, neverallow rules.

permissionx_id

A single named or anonymous permissionx. Used for allowx, auditallowx, dontauditx, neverallowx rules.

allow

Specifies the access allowed between a source and target type. Note that access may be refined by constraint rules based on the source, target and class (`validatetrans` or `mlsvalidatetrans`) or source, target class and permissions (`constrain` or `mlsconstrain` statements).

Rule definition:

```
(allow source_id target_id|self|notself|other classpermissionset_id ...)
```

Examples:

These examples show a selection of possible permutations of `allow` rules:

```
(class binder (impersonate call set_context_mgr transfer receive))
(class property_service (set))
(class zygote (specifyids specifyrlimits specifycapabilities specifyinvokewith specifyse

(classpermission cps_zygote)
(classpermissionset cps_zygote (zygote (not (specifyids))))

(classmap android_classes (set_1 set_2 set_3))

(classmapping android_classes set_1 (binder (all)))
(classmapping android_classes set_1 (property_service (set)))
(classmapping android_classes set_1 (zygote (not (specifycapabilities))))

(classmapping android_classes set_2 (binder (impersonate call set_context_mgr transfer))
(classmapping android_classes set_2 (zygote (specifyids specifyrlimits specifycapabiliti

(classmapping android_classes set_3 cps_zygote)
(classmapping android_classes set_3 (binder (impersonate call set_context_mgr)))

(block av_rules
    (type type_1)
    (type type_2)
    (type type_3)
    (type type_4)
    (type type_5)

    (typeattribute all_types)
    (typeattributeset all_types (all))

; These examples have named and anonymous classpermissionset's and
; classmap/classmapping statements
```

```
    (allow type_1 self (property_service (set)))          ; anonymous
    (allow type_2 self (zygote (specifyids)))             ; anonymous
    (allow type_3 self cps_zygote)                        ; named
    (allow type_4 self (android_classes (set_3)))         ; classmap/classmapping
    (allow all_types all_types (android_classes (set_2))) ; classmap/classmapping

;; This rule will cause the build to fail unless --disable-neverallow
;    (neverallow type_5 all_types (property_service (set)))
    (allow type_5 type_5 (property_service (set)))
    (allow type_1 all_types (property_service (set)))
)
```

auditallow

Audit the access rights defined if there is a valid allow rule. Note: It does NOT allow access, it only audits the event.

Rule definition:

```
(auditallow source_id target_id|self|notself|other classpermissionset_id)
```

Example:

This example will log an audit event whenever the corresponding `allow` rule grants access to the specified permissions:

```
(allow release_app.process secmark_demo.browser_packet (packet (send recv append bind)))

(auditallow release_app.process secmark_demo.browser_packet (packet (send recv)))
```

dontaudit

Do not audit the access rights defined when access denied. This stops excessive log entries for known events.

Note that these rules can be omitted by the CIL compiler command line parameter `-D` or `--disable-dontaudit` flags.

Rule definition:

```
(dontaudit source_id target_id|self|notself|other classpermissionset_id ...)
```

Example:

This example will not audit the denied access:

```
(dontaudit zygote.process self (capability (fsetid)))
```

neverallow

Never allow access rights defined. This is a compiler enforced action that will stop compilation until the offending rules are modified.

Note that these rules can be over-ridden by the CIL compiler command line parameter `-N` or `--disable-neverallow` flags.

Rule definition:

```
(neverallow source_id target_id|self|notself|other classpermissionset_id ...)
```

Example:

This example will not compile as `type_3` is not allowed to be a source type for the `allow` rule:

```
(class property_service (set))

(block av_rules
    (type type_1)
    (type type_2)
    (type type_3)
    (typeattribute all_types)
    (typeattributeset all_types (all))

    (neverallow type_3 all_types (property_service (set)))
    ; This rule will fail compilation:
    (allow type_3 self (property_service (set)))
)
```

deny

Remove the access rights defined from any matching allow rules. These rules are processed before `neverallow` checking.

Rule definition:

```
(deny source_id target_id|self classpermissionset_id ...)
```

Example:

```
(class class1 (perm1 perm2))

(type type1)
```

```
(type type2)
(allow type1 type2 (class1 (perm1))) ; Allow-1
(deny type1 type2 (class1 (perm1)))  ; Deny-1
; Allow-1 will be complete removed by Deny-1.

(type type3)
(type type4)
(allow type3 type4 (class1 (perm1 perm2))) ; Allow-2
(deny type3 type4 (class1 (perm1)))        ; Deny-2
; Allow-2 will be removed and replaced with the following when Deny-2 is evaluated
; (allow type3 type4 (class1 (perm2)))

(type type5)
(type type6)
(typeattribute attr1)
(typeattributeset attr1 (type5 type6))
(allow attr1 attr1 (class1 (perm1))) ; Allow-3
(deny type5 type6 (class1 (perm1)))  ; Deny-3
; Allow-3 will be removed and replaced with the following when Deny-3 is evaluated
; (allow type6 attr1 (class1 (perm1)))
; (allow type5 type5 (class1 (perm1)))
```

allowx

Specifies the access allowed between a source and target type using extended permissions. Unlike the `allow` statement, the statements `validatetrans`, `mlsvalidatetrans`, `constrain`, and `mlsconstrain` do not limit accesses granted by `allowx`.

Note that for this to work there must *also* be valid equivalent `allow` rules present.

Rule definition:

```
(allowx source_id target_id|self|notself|other permissionx_id)
```

Examples:

These examples show a selection of possible permutations of `allowx` rules:

```
(allow type_1 type_2 (tcp_socket (ioctl))) ;; pre-requisite
(allowx type_1 type_2 (ioctl tcp_socket (range 0x2000 0x20FF)))

(permissionx ioctl_nodebug (ioctl udp_socket (not (range 0x4000 0x4010))))
(allow type_3 type_4 (udp_socket (ioctl))) ;; pre-requisite
(allowx type_3 type_4 ioctl_nodebug)
```

auditallowx

Audit the access rights defined if there is a valid `allowx` rule. It does NOT allow access, it only audits the event.

Note that for this to work there must *also* be valid equivalent `auditallow` rules present.

Rule definition:

```
(auditallowx source_id target_id|self|notself|other permissionx_id)
```

Examples:

This example will log an audit event whenever the corresponding `allowx` rule grants access to the specified extended permissions:

```
(allowx type_1 type_2 (ioctl tcp_socket (range 0x2000 0x20FF)))

(auditallow type_1 type_2 (tcp_socket (ioctl))) ;; pre-requisite
(auditallowx type_1 type_2 (ioctl tcp_socket (range 0x2005 0x2010)))
```

dontauditx

Do not audit the access rights defined when access denied. This stops excessive log entries for known events.

Note that for this to work there must *also* be at least one `allowx` rule associated with the target type.

Note that these rules can be omitted by the CIL compiler command line parameter `-D` or `--disable-dontaudit` flags.

Rule definition:

```
(dontauditx source_id target_id|self|notself|other permissionx_id)
```

Examples:

This example will not audit the denied access:

```
(allowx type_1 type_2 (ioctl tcp_socket (0x1))) ;; pre-requisite, just some irrelevant
(dontauditx type_1 type_2 (ioctl tcp_socket (range 0x3000 0x30FF)))
```

neverallowx

Never allow access rights defined for extended permissions. This is a compiler enforced action that will stop compilation until the offending rules are modified.

Note that these rules can be over-ridden by the CIL compiler command line parameter `-N` or `--disable-neverallow` flags.

Rule definition:

```
(neverallowx source_id target_id|self|notself|other permissionx_id)
```

Examples:

This example will not compile as `type_3` is not allowed to be a source type and ioctl range for the `allowx` rule:

```
(class property_service (ioctl))
(block av_rules
    (type type_1)
    (type type_2)
    (type type_3)
    (typeattribute all_types)
    (typeattributeset all_types (all))
    (neverallowx type_3 all_types (ioctl property_service (range 0x2000 0x20FF)))
    ; This rule will fail compilation:
    (allowx type_3 self (ioctl property_service (0x20A0)))
)
```

Call / Macro Statements

call

Instantiate a macro within the current namespace. There may be zero or more parameters passed to the macro (with zero parameters this is similar to the `blockinherit` (`call`) / `blockabstract` (`macro`) statements).

Each parameter passed contains an argument to be resolved by the macro, these can be named or anonymous but must conform to the parameter types defined in the `macro` statement.

Macro rules are resolved by searching in the following order:

- The macro namespace (If found this means that the name was declared in the macro and is now declared in the namespace of one of the parents of the call.)

- The call arguments
- The parent namespaces of the macro being called (if any) with the exception of the global namespace.
- The parent namespaces of the call (if any) with the exception of the global namespace.
- The global namespace

Statement definition:

```
(call macro_id [(param ...)])
```

Where:

call

The call keyword.

macro_id

The identifier of the macro to be instantiated.

param

Zero or more parameters that are passed to the macro.

Example:

See the `macro` statement for an example.

macro

Declare a macro in the current namespace with its associated parameters. The macro identifier is used by the `call` statement to instantiate the macro and resolve any parameters. The call statement may be within the body of a macro.

`tunable`, `in`, `block`, `blockinherit`, `blockabstract`, and other `macro` statements are not allowed in `macro` blocks.

Duplicate `macro` declarations in the same namespace will normally cause an error, but inheriting a macro into a namespace (with `blockinherit`) that already has a macro with the same name will only result in a warning message and not cause an error. This behavior allows inherited macros to be overridden with local ones.

Statement definition:

```
(macro macro_id ([(param_type param_id) ...])
    cil_statements
    ...
)
```

Where:

macro

The macro keyword.

macro_id

The macro identifier.

param_type

Zero or more parameters that are passed to the macro. The param_type is a keyword used to determine the declaration type (e.g. type, class, categoryset).

The list of valid param_type entries are: string, name, type, role, user, sensitivity, category, bool, categoryset, level, levelrange, ipaddr, class, classmap, and classpermission.

The param_types categoryset, level, levelrange, classpermission, and ipaddr can by named or anonymous.

The param_types type, role, and user can be used for attributes.

The param_types type, sensitivity and category can be used for aliases.

The param_types name and string can be used interchangeably for an object_name in `typetransition` and the path in `filecon` statements.

param_id

The parameter identifier used to reference the entry within the macro body (e.g. ARG1).

cil_statement

Zero or more valid CIL statements.

Examples:

This example will instantiate the `binder_call` macro in the calling namespace (`my_domain`) and replace `ARG1` with `appdomain` and `ARG2` with `binderservicedomain`:

```
(block my_domain
    (call binder_call (appdomain binderservicedomain))
)

(macro binder_call ((type ARG1) (type ARG2))
```

```
    (allow ARG1 ARG2 (binder (call transfer)))
    (allow ARG2 ARG1 (binder (transfer)))
    (allow ARG1 ARG2 (fd (use)))
)
```

This example does not pass any parameters to the macro but adds a `type` identifier to the current namespace:

```
(block unconfined
    (call add_type)
    ....

    (macro add_type ()
        (type exec)
    )
)
```

This example passes an anonymous and named IP address to the macro:

```
(ipaddr netmask_1 255.255.255.0)
(context netlabel_1 (system.user object_r unconfined.object low_low))

(call build_nodecon ((192.168.1.64) netmask_1))

(macro build_nodecon ((ipaddr ARG1) (ipaddr ARG2))
    (nodecon ARG1 ARG2  netlabel_1)
)
```

Class and Permission Statements

common

Declares a common identifier in the current namespace with a set of common permissions that can be used by one or more `class` identifiers. The `classcommon` statement is used to associate a `common` identifier to a specific `class` identifier.

Statement definition:

```
(common common_id (permission_id ...))
```

Where:

common

The common keyword.

common_id

The common identifier.

permission_id

One or more permissions.

Example:

This common statement will associate the `common` identifier '`file`' with the list of permissions:

```
(common file (ioctl read write create getattr setattr lock relabelfrom relabelto append
```

classcommon

Associate a `class` identifier to a one or more permissions declared by a `common` identifier.

Statement definition:

```
(classcommon class_id common_id)
```

Where:

classcommon

The classcommon keyword.

class_id

A single previously declared class identifier.

common_id

A single previously declared common identifier that defines the common permissions for that class.

Example:

This associates the `dir` class with the list of permissions declared by the `file common` identifier:

```
(common file (ioctl read write create getattr setattr lock relabelfrom relabelto append

(classcommon dir file)
```

class

Declares a class and zero or more permissions in the current namespace.

Statement definition:

```
(class class_id (permission_id ...))
```

Where:

class

The class keyword.

class_id

The class identifier.

permission_id

Zero or more permissions declared for the class. Note that if zero permissions, an empty list is required as shown in the example.

Examples:

This example defines a set of permissions for the `binder` class identifier:

```
(class binder (impersonate call set_context_mgr transfer receive))
```

This example defines a common set of permissions to be used by the `sem` class, the `(class sem ())` does not define any other permissions (i.e. an empty list):

```
(common ipc (create destroy getattr setattr read write associate unix_read unix_write))

(classcommon sem ipc)
(class sem ())
```

and will produce the following set of permissions for the `sem` class identifier of:

```
(class sem (create destroy getattr setattr read write associate unix_read unix_write))
```

This example, with the following combination of the `common`, `classcommon` and `class` statements:

```
(common file (ioctl read write create getattr setattr lock relabelfrom relabelto append

(classcommon dir file)
(class dir (add_name remove_name reparent search rmdir open audit_access execmod))
```

will produce a set of permissions for the `dir` class identifier of:

```
(class dir (add_name remove_name reparent search rmdir open audit_access execmod ioctl r
```

classorder

Defines the order of class's. This is a mandatory statement. Multiple `classorder` statements declared in the policy will form an ordered list.

Statement definition:

```
(classorder (class_id ...))
```

Where:

classorder

The classorder keyword.

class_id

One or more class identifiers.

Example:

This will produce an ordered list of "`file dir process`"

```
(class process)
(class file)
(class dir)
(classorder (file dir))
(classorder (dir process))
```

Unordered Classorder Statement:

If users do not have knowledge of the existing `classorder`, the `unordered` keyword may be used in a `classorder` statement. The classes in an unordered statement are appended to the existing `classorder`. A class in an ordered statement always supersedes the class redeclaration in an unordered statement. The `unordered` keyword must be the first item in the `classorder` listing.

Example:

This will produce an unordered list of "`file dir foo a bar baz`"

```
(class file)
(class dir)
(class foo)
(class bar)
(class baz)
(class a)
(classorder (file dir))
(classorder (dir foo))
(classorder (unordered a))
(classorder (unordered bar foo baz))
```

classpermission

Declares a class permission set identifier in the current namespace that can be used by one or more `classpermissionset`s to associate one or more classes and permissions to form a named set.

Statement definition:

```
(classpermission classpermissionset_id)
```

Where:

classpermission

The classpermission keyword.

classpermissionset_id

The classpermissionset identifier.

Example:

See the `classpermissionset` statement for examples.

classpermissionset

Defines a class permission set identifier in the current namespace that associates a class and one or more permissions to form a named set. Nested expressions may be used to determine the required permissions as shown in the examples. Anonymous `classpermissionset`s may be used in av rules and constraints.

Statement definition:

```
(classpermissionset classpermissionset_id (class_id (permission_id | expr ...)))
```

Where:

classpermissionset

The classpermissionset keyword.

classpermissionset_id

The classpermissionset identifier.

class_id

A single previously declared class identifier.

permission_id

Zero or more permissions required by the class.

Note that there must be at least one permission identifier or expr declared).

expr

Zero or more expr's, the valid operators and syntax are:

(and (permission_id ...) (permission_id ...))

(or (permission_id ...) (permission_id ...))

(xor (permission_id ...) (permission_id ...))

(not (permission_id ...))

(all)

Examples:

These class permission set statements will resolve to the permission sets shown in the kernel policy language `allow` rules:

```
(class zygote (specifyids specifyrlimits specifycapabilities specifyinvokewith specifyse

(type test_1)
(type test_2)
(type test_3)
(type test_4)
(type test_5)

; NOT
(classpermission zygote_1)
(classpermissionset zygote_1 (zygote
    (not
        (specifyinvokewith specifyseinfo)
    )
))
(allow unconfined.process test_1 zygote_1)
;; allow unconfined.process test_1 : zygote { specifyids specifyrlimits specifycapabili

; AND - ALL - NOT - Equiv to test_1
(classpermission zygote_2)
(classpermissionset zygote_2 (zygote
    (and
        (all)
        (not (specifyinvokewith specifyseinfo))
    )
))
(allow unconfined.process test_2 zygote_2)
;; allow unconfined.process test_2 : zygote { specifyids specifyrlimits specifycapabili
```

```
; OR
(classpermission zygote_3)
(classpermissionset zygote_3 (zygote ((or (specifyinvokewith) (specifyseinfo)))))
(allow unconfined.process test_3 zygote_3)
;; allow unconfined.process test_3 : zygote { specifyinvokewith specifyseinfo } ;

; XOR - This will not produce an allow rule as the XOR will remove all the permissions:
(classpermission zygote_4)
(classpermissionset zygote_4 (zygote (xor (specifyids specifyrlimits specifycapabilities

; ALL
(classpermission zygote_all_perms)
(classpermissionset zygote_all_perms (zygote (all)))
(allow unconfined.process test_5 zygote_all_perms)
;; allow unconfined.process test_5 : zygote { specifyids specifyrlimits specifycapabili
```

classmap

Declares a class map identifier in the current namespace and one or more class mapping identifiers. This will allow:

1. Multiple `classpermissionset`s to be linked to a pair of `classmap` / `classmapping` identifiers.
2. Multiple `classs` to be associated to statements and rules that support a list of classes:

   typetransition typechange typemember rangetransition roletransition defaultuser defaultrole defaulttype defaultrange validatetrans mlsvalidatetrans

Statement definition:

```
(classmap classmap_id (classmapping_id ...))
```

Where:

classmap

The classmap keyword.

classmap_id

The classmap identifier.

classmapping_id

One or more classmapping identifiers.

Example:

See the `classmapping` statement for examples.

classmapping

Define sets of `classpermissionset`s (named or anonymous) to form a consolidated `classmapping` set. Generally there are multiple `classmapping` statements with the same `classmap` and `classmapping` identifiers that form a set of different `classpermissionset`'s. This is useful when multiple class / permissions are required in rules such as the `allow` rules (as shown in the examples).

Statement definition:

```
(classmapping classmap_id classmapping_id classpermissionset_id)
```

Where:

classmapping

The classmapping keyword.

classmap_id

A single previously declared classmap identifier.

classmapping_id

The classmapping identifier.

classpermissionset_id

A single named classpermissionset identifier or a single anonymous classpermissionset using expr's as required (see the classpermissionset statement).

Examples:

These class mapping statements will resolve to the permission sets shown in the kernel policy language `allow` rules:

```
(class binder (impersonate call set_context_mgr transfer receive))
(class property_service (set))
(class zygote (specifyids specifyrlimits specifycapabilities specifyinvokewith specifyse

(classpermission cps_zygote)
(classpermissionset cps_zygote (zygote (not (specifyids))))

(classmap android_classes (set_1 set_2 set_3))

(classmapping android_classes set_1 (binder (all)))
```

```
(classmapping android_classes set_1 (property_service (set)))
(classmapping android_classes set_1 (zygote (not (specifycapabilities))))

(classmapping android_classes set_2 (binder (impersonate call set_context_mgr transfer)))
(classmapping android_classes set_2 (zygote (specifyids specifyrlimits specifycapabiliti

(classmapping android_classes set_3 cps_zygote)
(classmapping android_classes set_3 (binder (impersonate call set_context_mgr)))

(block map_example
    (type type_1)
    (type type_2)
    (type type_3)

    (allow type_1 self (android_classes (set_1)))
    (allow type_2 self (android_classes (set_2)))
    (allow type_3 self (android_classes (set_3)))
)

; The above will resolve to the following AV rules:
;; allow map_example.type_1 map_example.type_1 : binder { impersonate call set_context_
;; allow map_example.type_1 map_example.type_1 : property_service set ;
;; allow map_example.type_1 map_example.type_1 : zygote { specifyids specifyrlimits spec

;; allow map_example.type_2 map_example.type_2 : binder { impersonate call set_context_
;; allow map_example.type_2 map_example.type_2 : zygote { specifyids specifyrlimits spec

;; allow map_example.type_3 map_example.type_3 : binder { impersonate call set_context_
;; allow map_example.type_3 map_example.type_3 : zygote { specifyrlimits specifycapabil
```

permissionx

Defines a named extended permission, which can be used in the `allowx`, `auditallowx`, `dontauditx`, and `neverallowx` statements.

Statement definition:

```
(permissionx permissionx_id (kind class_id (permission ... | expr ...)))
```

Where:

permissionx

The permissionx keyword.

kind

A keyword specifying how to interpret the extended permission values. Must be one of:

kind

description

ioctl

Permissions define a whitelist of ioctl command values. Permission values must range from 0x0000 to 0xFFFF, inclusive.

nlmsg

Permissions define a whitelist of netlink message type values. Permission values must range from 0x0000 to 0xFFFF, inclusive.

class_id

A single previously declared class or classmap identifier.

permission

One or more numeric values, specified in decimal, or hexadecimal if prefixed with 0x, or octal if prefixed with 0. Values are interpreted based on the value of kind.

expr

An expression, with valid operators and syntax:

(range (permission ...) (permission ...))

(and (permission ...) (permission ...))

(or (permission ...) (permission ...))

(xor (permission ...) (permission ...))

(not (permission ...))

(all)

Examples:

```
(permissionx ioctl_1 (ioctl tcp_socket (0x2000 0x3000 0x4000)))
(permissionx ioctl_2 (ioctl tcp_socket (range 0x6000 0x60FF)))
(permissionx ioctl_3 (ioctl tcp_socket (and (range 0x8000 0x90FF) (not (range 0x8100 0x8
(permissionx nlmsg_1 (nlmsg netlink_route_socket (range 0x0010 0x0018)))
```

Conditional Statements

boolean

Declares a run time boolean as true or false in the current namespace. The `booleanif` statement contains the CIL code that will be in the binary policy file.

`boolean` are not allowed in `booleanif` blocks.

Statement definition:

```
(boolean boolean_id true|false)
```

Where:

boolean

The boolean keyword.

boolean_id

The boolean identifier.

true | false

The initial state of the boolean. This can be changed at run time using setsebool(8) and its status queried using getsebool(8).

Example:

See the `booleanif` statement for an example.

booleanif

Contains the run time conditional statements that are instantiated in the binary policy according to the computed boolean identifier(s) state.

`call` statements are allowed within a `booleanif`, however the contents of the resulting macro must be limited to those of the `booleanif` statement (i.e. `allow`, `auditallow`, `dontaudit`, `typemember`, `typetransition`, `typechange` and the compile time `tunableif` statement)).

Statement definition:

```
(booleanif boolean_id | expr ...
    (true
        cil_statements
        ...)
    (false
```

```
        cil_statements
        ...)
)
```

Where:

booleanif

The booleanif keyword.

boolean_id

Either a single boolean identifier or one or more expr's.

expr

Zero or more expr's, the valid operators and syntax are:

(and boolean_id boolean_id)

(or boolean_id boolean_id)

(xor boolean_id boolean_id)

(eq boolean_id boolean_id)

(neq boolean_id boolean_id)

(not boolean_id)

true

An optional set of CIL statements that will be instantiated when the boolean is evaluated as true.

false

An optional set of CIL statements that will be instantiated when the boolean is evaluated as false.

Examples:

The second example also shows the kernel policy language equivalent:

```
(boolean disableAudio false)

(booleanif disableAudio
    (false
        (allow process mediaserver.audio_device (chr_file_set (rw_file_perms)))
    )
)

(boolean disableAudioCapture false)
```

```
;;; if(!disableAudio && !disableAudioCapture) {
(booleanif (and (not disableAudio) (not disableAudioCapture))
    (true
        (allow process mediaserver.audio_capture_device (chr_file_set (rw_file_perms)))
    )
)
```

tunable

Tunables are similar to booleans, however they are used to manage areas of CIL statements that may or may not be in the final CIL policy that will be compiled (whereas booleans are embedded in the binary policy and can be enabled or disabled during run-time).

Note that tunables can be treated as booleans by the CIL compiler command line parameter `-P` or `--preserve-tunables` flags.

Since `tunableif` statements are resolved first, `tunable` statements are not allowed in `in`, `macro`, `optional`, and `booleanif` blocks. To simplify processing, they are also not allowed in `tunableif` blocks.

Statement definition:

```
(tunable tunable_id true|false)
```

Where:

tunable

The tunable keyword.

tunable_id

The tunable identifier.

true | false

The initial state of the tunable.

Example:

See the `tunableif` statement for an example.

tunableif

Compile time conditional statement that may or may not add CIL statements to be compiled.

If tunables are being treated as booleans (by using the CIL compiler command line parameter `-P` or `--preserve-tunables` flag), then only the statements

allowed in a `booleanif` block are allowed in a `tunableif` block. Otherwise, `tunable` statements are not allowed in a `tunableif` block.

Statement definition:

```
(tunableif tunable_id | expr ...
    (true
        cil_statements
        ...)
    (false
        cil_statements
        ...)
)
```

Where:

tunableif

The tunableif keyword.

tunable_id

Either a single tunable identifier or one or more expr's.

expr

Zero or more expr's, the valid operators and syntax are:

(and tunable_id tunable_id)

(or tunable_id tunable_id)

(xor tunable_id tunable_id)

(eq tunable_id tunable_id)

(neq tunable_id tunable_id)

(not tunable_id)

true

An optional set of CIL statements that will be instantiated when the tunable is evaluated as true.

false

An optional set of CIL statements that will be instantiated when the tunable is evaluated as false.

Example:

This example will not add the range transition rule to the binary policy:

```
(tunable range_trans_rule false)

(block init
    (class process (process))
    (type process)

    (tunableif range_trans_rule
        (true
            (rangetransition process sshd.exec process low_high)
        )
    ) ; End tunableif
) ; End block
```

Constraint Statements

constrain

Enable constraints to be placed on the specified permissions of the object class based on the source and target security context components.

Statement definition:

```
(constrain classpermissionset_id ... expression | expr ...)
```

Where:

constrain

The constrain keyword.

classpermissionset_id

A single named or anonymous classpermissionset or a single set of classmap/classmapping identifiers.

expression

There must be one constraint expression or one or more expr's. The expression consists of an operator and two operands as follows:

(op u1 u2)

(role_op r1 r2)

(op t1 t2)

(op u1 user_id | (user_id ...))

(op u2 user_id | (user_id ...))

(op r1 role_id | (role_id ...))

(op r2 role_id | (role_id ...))

(op t1 type_id | (type_id ...))

(op t2 type_id | (type_id ...))

where:

u1, r1, t1 = Source context: user, role or type

u2, r2, t2 = Target context: user, role or type

and:

op : eq neq

role_op : eq neq dom domby incomp

user_id : A single user or userattribute identifier.

role_id : A single role or roleattribute identifier.

type_id : A single type, typealias or typeattribute identifier.

expr

Zero or more expr's, the valid operators and syntax are:

(and expression expression)

(or expression expression)

(not expression)

Examples:

Two constrain statements are shown with their equivalent kernel policy language statements:

```
;; constrain { file } { write }
;;    (( t1 == unconfined.process  ) and ( t2 == unconfined.object  ) or ( r1 eq r2 ));
(constrain (file (write))
    (or
        (and
            (eq t1 unconfined.process)
            (eq t2 unconfined.object)
        )
        (eq r1 r2)
    )
)

;; constrain { file } { read }
;;    (not( t1 == unconfined.process  ) and ( t2 == unconfined.object  ) or ( r1 eq r2 )
```

```
(constrain (file (read))
    (not
        (or
            (and
                (eq t1 unconfined.process)
                (eq t2 unconfined.object)
            )
            (eq r1 r2)
        )
    )
)
```

validatetrans

The `validatetrans` statement is only used for `file` related object classes where it is used to control the ability to change the objects security context based on old, new and the current process security context.

Statement definition:

```
(validatetrans class_id expression | expr ...)
```

Where:

validatetrans

The validatetrans keyword.

class_id

A single previously declared class or classmap identifier.

expression

There must be one constraint expression or one or more expr's. The expression consists of an operator and two operands as follows:

(op u1 u2)

(role_op r1 r2)

(op t1 t2)

(op u1 user_id)

(op u2 user_id)

(op u3 user_id)

(op r1 role_id)

(op r2 role_id)

(op r3 role_id)

(op t1 type_id)

(op t2 type_id)

(op t3 type_id)

where:

u1, r1, t1 = Old context: user, role or type

u2, r2, t2 = New context: user, role or type

u3, r3, t3 = Process context: user, role or type

and:

op : eq neq

role_op : eq neq dom domby incomp

user_id : A single user or userattribute identifier.

role_id : A single role or roleattribute identifier.

type_id : A single type, typealias or typeattribute identifier.

expr

Zero or more expr's, the valid operators and syntax are:

(and expression expression)

(or expression expression)

(not expression)

Example:

A validate transition statement with the equivalent kernel policy language statement:

```
; validatetrans { file } ( t1 == unconfined.process  );
```

```
(validatetrans file (eq t1 unconfined.process))
```

mlsconstrain

Enable MLS constraints to be placed on the specified permissions of the object class based on the source and target security context components.

Statement definition:

```
(mlsconstrain classpermissionset_id ... expression | expr ...)
```

Where:

mlsconstrain

The mlsconstrain keyword.

classpermissionset_id

A single named or anonymous classpermissionset or a single set of classmap/classmapping identifiers.

expression

There must be one constraint expression or one or more expr's. The expression consists of an operator and two operands as follows:

(op u1 u2)

(mls_role_op r1 r2)

(op t1 t2)

(mls_role_op l1 l2)

(mls_role_op l1 h2)

(mls_role_op h1 l2)

(mls_role_op h1 h2)

(mls_role_op l1 h1)

(mls_role_op l2 h2)

(op u1 user_id)

(op u2 user_id)

(op r1 role_id)

(op r2 role_id)

(op t1 type_id)

(op t2 type_id)

where:

u1, r1, t1, l1, h1 = Source context: user, role, type, low level or high level

u2, r2, t2, l2, h2 = Target context: user, role, type, low level or high level

and:

op : eq neq

mls_role_op : eq neq dom domby incomp

user_id : A single user or userattribute identifier.

role_id : A single role or roleattribute identifier.

type_id : A single type, typealias or typeattribute identifier.

expr

Zero or more expr's, the valid operators and syntax are:

(and expression expression)

(or expression expression)

(not expression)

Example:

An MLS constrain statement with the equivalent kernel policy language statement:

```
;; mlsconstrain { file } { open }
;;     (( l1 eq l2 ) and ( u1 == u2 ) or ( r1 != r2 ));

(mlsconstrain (file (open))
    (or
        (and
            (eq l1 l2)
            (eq u1 u2)
        )
        (neq r1 r2)
    )
)
```

mlsvalidatetrans

The `mlsvalidatetrans` statement is only used for `file` related object classes where it is used to control the ability to change the objects security context based on old, new and the current process security context.

Statement definition:

```
(mlsvalidatetrans class_id expression | expr ...)
```

Where:

mlsvalidatetrans

The mlsvalidatetrans keyword.

class_id

A single previously declared class or classmap identifier.

expression

There must be one constraint expression or one or more expr's. The expression consists of an operator and two operands as follows:

(op u1 u2)

(mls_role_op r1 r2)

(op t1 t2)

(mls_role_op l1 l2)

(mls_role_op l1 h2)

(mls_role_op h1 l2)

(mls_role_op h1 h2)

(mls_role_op l1 h1)

(mls_role_op l2 h2)

(op u1 user_id)

(op u2 user_id)

(op u3 user_id)

(op r1 role_id)

(op r2 role_id)

(op r3 role_id)

(op t1 type_id)

(op t2 type_id)

(op t3 type_id)

where:

u1, r1, t1, l1, h1 = Source context: user, role, type, low level or high level

u2, r2, t2, l2, h2 = Target context: user, role, type, low level or high level

u3, r3, t3 = Process context: user, role or type

and:

op : eq neq

mls_role_op : eq neq dom domby incomp

user_id : A single user or userattribute identifier.

role_id : A single role or roleattribute identifier.

type_id : A single type, typealias or typeattribute identifier.

expr

Zero or more expr's, the valid operators and syntax are:

(and expression expression)

(or expression expression)

(not expression)

Example:

An MLS validate transition statement with the equivalent kernel policy language statement:

```
;; mlsvalidatetrans { file } ( l1 domby h2 );
```

```
(mlsvalidatetrans file (domby l1 h2))
```

Container Statements

block

Start a new namespace.

Not allowed in `macro` and `optional` blocks.

`sensitivity` and `category` statements are not allowed in `block` blocks.

Duplicate declarations of a `block` in the same namespace will normally cause an error, but inheriting a block into a namespace (with `blockinherit`) that already has a block with the same name will only result in a warning message and not cause an error. The policy from both blocks will end up in the binary policy. This behavior was used in the past to allow a block to be declared so that an `in-statement` could be used on it, but now an `in-statement` can be specified to occur after inheritance, so this behavior is not necessary (but is still allowed).

Statement definition:

```
(block block_id
    cil_statement
    ...
)
```

Where:

block

The block keyword.

block_id

The namespace identifier.

cil_statement

Zero or more valid CIL statements.

Example:

See the `blockinherit` statement for an example.

blockabstract

Declares the namespace as a 'template' and does not generate code until instantiated by another namespace that has a `blockinherit` statement.

Not allowed in `macro` and `optional` blocks.

Statement definition:

```
(block block_id
    (blockabstract template_id)
    cil_statement
    ...
)
```

Where:

block

The block keyword.

block_id

The namespace identifier.

blockabstract

The blockabstract keyword.

template_id

The abstract namespace identifier. This must match the block_id entry.

cil_statement

Zero or more valid CIL statements forming the abstract block.

Example:

See the `blockinherit` statement for an example.

blockinherit

Used to add common policy rules to the current namespace via a template that has been defined with the `blockabstract` statement. All `blockinherit` statements are resolved first and then the contents of the block are copied. This is so that inherited blocks will not be inherited. For a concrete example, please see the examples section.

Inherited rules are resolved by searching namespaces in the following order:

- The parent namespaces (if any) where the `blockinherit` rule is located with the exception of the global namespace.
- The parent namespaces of the block being inherited (but not that block's namespace) with the exception of the global namespace.
- The global namespace.

Not allowed in `macro` blocks.

Statement definition:

```
(block block_id
    (blockinherit template_id)
    cil_statement
    ...
)
```

Where:

block

The block keyword.

block_id

The namespace identifier.

blockinherit

The blockinherit keyword.

template_id

The inherited namespace identifier.

cil_statement

Zero or more valid CIL statements.

Example:

This example contains a template `client_server` that is instantiated in two blocks (`netserver_app` and `netclient_app`):

```
; This is the template block:
(block client_server
    (blockabstract client_server)

    ; Log file labeling
    (type log_file)
    (typeattributeset file_type (log_file))
    (typeattributeset data_file_type (log_file))
    (allow process log_file (dir (write search create setattr add_name)))
    (allow process log_file (file (create open append getattr setattr)))
    (roletype object_r log_file)
    (context log_file_context (u object_r log_file low_low))

    ; Process labeling
    (type process)
    (typeattributeset domain (process))
    (call app_domain (process))
    (call net_domain (process))
)

; This is a policy block that will inherit the abstract block above:
(block netclient_app
    ; Add common policy rules to namespace:
    (blockinherit client_server)
    ; Label the log files
    (filecon "/data/data/com.se4android.netclient/.*" file log_file_context)
)

; This is another policy block that will inherit the abstract block above:
(block netserver_app
   ; Add common policy rules to namespace:
    (blockinherit client_server)

    ; Label the log files
    (filecon "/data/data/com.se4android.netserver/.*" file log_file_context)
)

; This is an example of how blockinherits resolve inherits before copying
(block a
    (type one))

(block b
    ; Notice that block a is declared here as well
    (block a
        (type two)))
```

```
; This will first copy the contents of block b, which results in type b.a.two being cop
; Next, the contents of block a will be copied which will result in type a.one.
(block ab
    (blockinherit b)
    (blockinherit a))
```

optional

Declare an `optional` namespace. All CIL statements in the optional block must be satisfied before instantiation in the binary policy.

Not allowed in `booleanif` blocks.

`tunable`, `in`, `block`, `blockabstract`, and `macro` statements are not allowed in `optional` blocks.

Statement definition:

```
(optional optional_id
    cil_statement
    ...
)
```

Where:

optional

The optional keyword.

optional_id

The optional namespace identifier.

cil_statement

Zero or more valid CIL statements.

Example:

This example will instantiate the optional block `ext_gateway.move_file` into policy providing all optional CIL statements can be resolved:

```
(block ext_gateway
    ......
    (optional move_file
        (typetransition process msg_filter.move_file.in_queue file msg_filter.move_file.
        (allow process msg_filter.move_file.in_queue (dir (read getattr write search add
        (allow process msg_filter.move_file.in_file (file (write create getattr)))
        (allow msg_filter.move_file.in_file unconfined.object (filesystem (associate)))
        (typetransition msg_filter.int_gateway.process msg_filter.move_file.out_queue fi
```

```
            msg_filter.move_file.out_file)
        (allow msg_filter.int_gateway.process msg_filter.move_file.out_queue (dir (read
        (allow msg_filter.int_gateway.process msg_filter.move_file.out_file (file (read
    ) ; End optional block

    .....
) ; End block
```

in

Allows the insertion of CIL statements into a named container (`block`, `optional` or `macro`). This insertion can be specified to occur either before or after block inheritance has been resolved.

Not allowed in `macro`, `booleanif`, and other `in` blocks.

`tunable` and `in` statements are not allowed in `in` blocks.

Statement definition:

```
(in [before|after] container_id
    cil_statement
    ...
)
```

Where:

in

The in keyword.

before|after

An optional value that specifies whether to process the `in` before or after block inheritance. If no value is specified, then the `in` will be processed before block inheritance.

container_id

A valid block, optional or macro namespace identifier.

cil_statement

Zero or more valid CIL statements.

Example:

This will add rules to the container named `system_server`:

```
(in system_server
    (dontaudit process secmark_demo.dns_packet (packet (send recv)))
    (allow process secmark_demo.dns_packet (packet (send recv)))
)
```

Context Statement

Contexts are formed using previously declared parameters and may be named or anonymous where:

- Named - The context is declared with a context identifier that is used as a reference.
- Anonymous - They are defined within the CIL labeling statement using user, role etc. identifiers.

Each type is shown in the examples.

context

Declare an SELinux security context identifier for labeling. The range (or current and clearance levels) MUST be defined whether the policy is MLS/MCS enabled or not.

Statement definition:

```
(context context_id (user_id role_id type_id levelrange_id)))
```

Where:

context

The context keyword.

context_id

The context identifier.

user_id

A single previously declared user identifier.

role_id

A single previously declared role identifier.

type_id

A single previously declared type or typealias identifier.

levelrange_id

A single previously declared levelrange identifier. This entry may also be defined by anonymous or named level, sensitivity, sensitivityalias, category, categoryalias or categoryset as discussed in the Multi-Level Security Labeling Statements section and shown in the examples.

Examples:

This example uses a named context definition:

```
(context runas_exec_context (u object_r exec low_low))

(filecon "/system/bin/run-as" file runas_exec_context)
```

to resolve/build a `file_contexts` entry of (assuming MLS enabled policy):

```
/system/bin/run-as  -- u:object_r:runas.exec:s0-s0
```

This example uses an anonymous context where the previously declared `user role type levelrange` identifiers are used to specify two `portcon` statements:

```
(portcon udp 1024 (test.user object_r test.process ((s0) (s1))))
(portcon tcp 1024 (test.user object_r test.process (system_low system_high)))
```

This example uses an anonymous context for the first and named context for the second in a `netifcon` statement:

```
(context netif_context (test.user object_r test.process ((s0 (c0)) (s1 (c0)))))

(netifcon eth04 (test.user object_r test.process ((s0 (c0)) (s1 (c0)))) netif_context)
```

Default Object Statements

These rules allow a default user, role, type and/or range to be used when computing a context for a new object. These require policy version 27 or 28 with kernels 3.5 or greater.

defaultuser

Allows the default user to be taken from the source or target context when computing a new context for the object `class` identifier. Requires policy version 27.

Statement definition:

```
(defaultuser class_id default)
```

Where:

defaultuser

The defaultuser keyword.

class_id

A single previously declared class or classmap identifier, or a list of previously declared class or classmap identifiers enclosed within parentheses.

default

A keyword of either source or target.

Example:

When creating new `binder`, `property_service`, `zygote` or `memprotect` objects the `user` component of the new security context will be taken from the `source` context:

```
(class binder (impersonate call set_context_mgr transfer receive))
(class property_service (set))
(class zygote (specifyids specifyrlimits specifycapabilities specifyinvokewith specifyse
(class memprotect (mmap_zero))

(classmap android_classes (android))
(classmapping android_classes android (binder (all)))
(classmapping android_classes android (property_service (set)))
(classmapping android_classes android (zygote (not (specifycapabilities))))

(defaultuser (android_classes memprotect) source)

; Will produce the following in the binary policy file:
;; default_user binder source;
;; default_user zygote source;
;; default_user property_service source;
;; default_user memprotect source;
```

defaultrole

Allows the default role to be taken from the source or target context when computing a new context for the object `class` identifier. Requires policy version 27.

```
(defaultrole class_id default)
```

Where:

defaultrole

The defaultrole keyword.

class_id

A single previously declared class or classmap identifier, or a list of previously declared class or classmap identifiers enclosed within parentheses.

default

A keyword of either source or target.

Example:

When creating new `binder`, `property_service` or `zygote` objects the `role` component of the new security context will be taken from the `target` context:

```
(class binder (impersonate call set_context_mgr transfer receive))
(class property_service (set))
(class zygote (specifyids specifyrlimits specifycapabilities specifyinvokewith specifyse

(defaultrole (binder property_service zygote) target)

; Will produce the following in the binary policy file:
;; default_role binder target;
;; default_role zygote target;
;; default_role property_service target;
```

defaulttype

Allows the default type to be taken from the source or target context when computing a new context for the object `class` identifier. Requires policy version 28.

Statement definition:

```
(defaulttype class_id default)
```

Where:

defaulttype

The defaulttype keyword.

class_id

A single previously declared class or classmap identifier, or a list of previously declared class or classmap identifiers enclosed within parentheses.

default

A keyword of either source or target.

Example:

When creating a new `socket` object, the `type` component of the new security context will be taken from the `source` context:

```
(defaulttype socket source)
```

defaultrange

Allows the default level or range to be taken from the source, target, or both contexts when computing a new context for the object `class` identifier. Requires policy version 27. glblub as the default requires policy version 32.

Statement definition:

```
(defaultrange class_id default <range>)
```

Where:

defaultrange

The defaultrange keyword.

class_id

A single previously declared class or classmap identifier, or a list of previously declared class or classmap identifiers enclosed within parentheses.

default

A keyword of either source, target, or glblub.

range

A keyword of either low, high, or low-high.

Example:

When creating a new `file` object, the appropriate `range` component of the new security context will be taken from the `target` context:

```
(defaultrange file target low-high)
```

MLS userspace object managers may need to compute the common parts of a range such that the object is created with the range common to the subject and containing object:

```
(defaultrange db_table glblub)
```

File Labeling Statements

filecon

Define entries for labeling files. The compiler will produce these entries in a file called `file_contexts(5)` by default in the `cwd`. The compiler option `[-f|--filecontext <filename>]` may be used to specify a different path or file name.

Statement definition:

```
(filecon "path" file_type context_id)
```

Where:

filecon

The filecon keyword.

path

A string representing the file path that may be in the form of a regular expression. The string must be enclosed within double quotes (e.g. "/this/is/a/path(/.*)?")

file_type

A single keyword representing a file type in the file_contexts file as follows:

keyword

file_contexts entry

file

–

dir

-d

char

-c

block

-b

socket

-s

pipe

-p

symlink

-l

any

no entry

context_id

The security context to be allocated to the file, which may be:

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

An empty context list represented by () can be used to indicate that matching files should not be re-labeled. This will be interpreted as «none» within the file_contexts(5) file.

Examples:

These examples use one named, one anonymous and one empty context definition:

```
(context runas_exec_context (u object_r exec low_low))

(filecon "/system/bin/run-as" file runas_exec_context)
(filecon "/dev/socket/wpa_wlan[0-9]" any (u object_r wpa.socket ((s0)(s0))))
(filecon "/data/local/mine" dir ())
```

to resolve/build `file_contexts` entries of (assuming MLS enabled policy):

```
/system/bin/run-as  -- u:object_r:runas.exec:s0
/dev/socket/wpa_wlan[0-9]   u:object_r:wpa.socket:s0
/data/local/mine -d <<none>>
```

fsuse

Label filesystems that support SELinux security contexts.

Statement definition:

```
(fsuse fstype fsname context_id)
```

Where:

fsuse

The fsuse keyword.

fstype

A single keyword representing the type of filesystem as follows:

task - For pseudo filesystems supporting task related services such as pipes and sockets.

trans - For pseudo filesystems such as pseudo terminals and temporary objects.

xattr - Filesystems supporting the extended attribute security.selinux. The labeling is persistent for filesystems that support extended attributes.

fsname

Name of the supported filesystem (e.g. ext4 or pipefs).

context_id

The security context to be allocated to the network interface.

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

The context identifiers are declared in the `file` namespace and the `fsuse` statements in the global namespace:

```
(block file
    (type labeledfs)
    (roletype object_r labeledfs)
    (context labeledfs_context (u object_r labeledfs low_low))

    (type pipefs)
    (roletype object_r pipefs)
    (context pipefs_context (u object_r pipefs low_low))
    ...
)

(fsuse xattr ex4 file.labeledfs_context)
(fsuse xattr btrfs file.labeledfs_context)

(fsuse task pipefs file.pipefs_context)
(fsuse task sockfs file.sockfs_context)

(fsuse trans devpts file.devpts_context)
(fsuse trans tmpfs file.tmpfs_context)
```

genfscon

Used to allocate a security context to filesystems that cannot support any of the `fsuse` file labeling options. Generally a filesystem would have a single default security context assigned by `genfscon` from the root `(/)` that would then be inherited by all files and directories on that filesystem. The exception to this is the `/proc` filesystem, where directories can be labeled with a specific security context (as shown in the examples).

Statement definition:

```
(genfscon fsname path [file_type] context_id)
```

Where:

genfscon

The genfscon keyword.

fsname

Name of the supported filesystem (e.g. rootfs or proc).

path

If fsname is proc, then the partial path (see examples). For all other types this must be '/'.

file_type

Optional keyword representing a file type. Valid values are the same as in `filecon` rules.

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

The context identifiers are declared in the `file` namespace and the `genfscon` statements are then inserted using the `in` container statement:

```
(file
    (type rootfs)
    (roletype object_r rootfs)
    (context rootfs_context (u object_r rootfs low_low))

    (type proc)
    (roletype object_r proc)
```

```
    (context rootfs_context (u object_r proc low_low))
    ...
)

(in file
    (genfscon rootfs / rootfs_context)
    ; proc labeling can be further refined (longest matching prefix).
    (genfscon proc / proc_context)
    (genfscon proc /net/xt_qtaguid/ctrl qtaguid_proc_context)
    (genfscon proc /sysrq-trigger sysrq_proc_context)
    (genfscon selinuxfs / selinuxfs_context)
)
```

Multi-Level Security Labeling Statements

Because there are many options for MLS labeling, the examples show a limited selection of statements, however there is a simple policy that will build shown in the `levelrange` section.

sensitivity

Declare a sensitivity identifier in the current namespace. Multiple `sensitivity` statements in the policy will form an ordered list.

Statement definition:

```
(sensitivity sensitivity_id)
```

Where:

sensitivity

The sensitivity keyword.

sensitivity_id

The sensitivity identifier.

Example:

This example declares three `sensitivity` identifiers:

```
(sensitivity s0)
(sensitivity s1)
(sensitivity s2)
```

sensitivityalias

Declares a sensitivity alias identifier in the current namespace. See the `sensitivityaliasactual` statement for an example that associates the `sensitivityalias` identifier.

Statement definition:

```
(sensitivityalias sensitivityalias_id)
```

Where:

sensitivityalias

The sensitivityalias keyword.

sensitivityalias_id

The sensitivityalias identifier.

Example:

See the `sensitivityaliasactual` statement.

sensitivityaliasactual

Associates a previously declared `sensitivityalias` identifier to a previously declared `sensitivity` identifier.

Statement definition:

```
(sensitivityaliasactual sensitivityalias_id sensitivity_id)
```

Where:

sensitivityaliasactual

The sensitivityaliasactual keyword.

sensitivityalias_id

A single previously declared sensitivityalias identifier.

sensitivity_id

A single previously declared sensitivity identifier.

Example:

This example will associate sensitivity `s0` with two sensitivity alias's:

```
(sensitivity s0)
(sensitivityalias unclassified)
(sensitivityalias SystemLow)
(sensitivityaliasactual unclassified s0)
(sensitivityaliasactual SystemLow s0)
```

sensitivityorder

Define the sensitivity order - lowest to highest. Multiple `sensitivityorder` statements in the policy will form an ordered list.

Statement definition:

```
(sensitivityorder (sensitivity_id ...))
```

Where:

sensitivityorder

The sensitivityorder keyword.

sensitivity_id

One or more previously declared sensitivity or sensitivityalias identifiers..

Example:

This example shows two `sensitivityorder` statements that when compiled will form an ordered list. Note however that the second `sensitivityorder` statement starts with `s2` so that the ordered list can be built.

```
(sensitivity s0)
(sensitivityalias s0 SystemLow)
(sensitivity s1)
(sensitivity s2)
(sensitivityorder (SystemLow s1 s2))

(sensitivity s3)
(sensitivity s4)
(sensitivityalias s4 SystemHigh)
(sensitivityorder (s2 s3 SystemHigh))
```

category

Declare a category identifier in the current namespace. Multiple category statements declared in the policy will form an ordered list.

Statement definition:

```
(category category_id)
```

Where:

category

The category keyword.

category_id

The category identifier.

Example:

This example declares a three `category` identifiers:

```
(category c0)
(category c1)
(category c2)
```

categoryalias

Declares a category alias identifier in the current namespace. See the `categoryaliasactual` statement for an example that associates the `categoryalias` identifier.

Statement definition:

```
(categoryalias categoryalias_id)
```

Where:

categoryalias

The categoryalias keyword.

categoryalias_id

The categoryalias identifier.

categoryaliasactual

Associates a previously declared `categoryalias` identifier to a previously declared `category` identifier.

Statement definition:

```
(categoryaliasactual categoryalias_id category_id)
```

Where:

categoryaliasactual

The categoryaliasactual keyword.

categoryalias_id

A single previously declared categoryalias identifier.

category_id

A single previously declared category identifier.

Example:

Declares a category `c0`, a category alias of `documents`, and then associates them:

```
(category c0)
(categoryalias documents)
(categoryaliasactual documents c0)
```

categoryorder

Define the category order. Multiple `categoryorder` statements declared in the policy will form an ordered list. Note that this statement orders the categories to allow validation of category ranges.

Statement definition:

```
(categoryorder (category_id ...))
```

Where:

categoryorder

The categoryorder keyword.

category_id

One or more previously declared category or categoryalias identifiers.

Example:

This example orders one category alias and nine categories:

```
(categoryorder (documents c1 c2 c3 c4 c5 c6 c7 c8 c9))
```

categoryset

Declare an identifier for a set of contiguous or non-contiguous categories in the current namespace.

Notes:

- Category expressions are allowed in `categoryset`, `sensitivitycategory`, `level`, and `levelrange` statements.
- Category sets are not allowed in `categoryorder` statements.

Statement definition:

```
(categoryset categoryset_id (category_id ... | expr ...))
```

Where:

categoryset

The categoryset keyword.

categoryset_id

The categoryset identifier.

category_id

Zero or more previously declared category or categoryalias identifiers.

Note that there must be at least one category_id identifier or expr parameter declared.

expr

Zero or more expr's, the valid operators and syntax are:

(and (category_id ...) (category_id ...))

(or (category_id ...) (category_id ...))

(xor (category_id ...) (category_id ...))

(not (category_id ...))

(range category_id category_id)

(all)

Examples:

These examples show a selection of `categoryset` statements:

```
; Declare categories with two alias's:
(category c0)
(categoryalias documents)
(categoryaliasactual documents c0)
(category c1)
(category c2)
(category c3)
(category c4)
(categoryalias spreadsheets)
(categoryaliasactual spreadsheets c4)

; Set the order to determine ranges:
(categoryorder (c0 c1 c2 c3 spreadsheets))

(categoryset catrange_1 (range c2 c3))

; Two methods to associate all categories:
(categoryset all_cats (range c0 c4))
(categoryset all_cats1 (all))

(categoryset catset_1 (documents c1))
(categoryset catset_2 (c2 c3))
(categoryset catset_3 (c4))

(categoryset just_c0 (xor (c1 c2) (documents c1 c2)))
```

sensitivitycategory

Associate a `sensitivity` identifier with one or more category's. Multiple definitions for the same `sensitivity` form an ordered list of categories for that sensitivity. This statement is required before a `level` identifier can be declared.

Statement definition:

```
(sensitivitycategory sensitivity_id categoryset_id)
```

Where:

sensitivitycategory

The sensitivitycategory keyword.

sensitivity_id

A single previously declared sensitivity or sensitivityalias identifier.

categoryset_id

A single previously declared categoryset (named or anonymous), or a list of category and/or categoryalias identifiers. The examples show each variation.

Examples:

These `sensitivitycategory` examples use a selection of `category`, `categoryalias` and `categoryset`'s:

```
(sensitivitycategory s0 catrange_1)
(sensitivitycategory s0 catset_1)
(sensitivitycategory s0 catset_3)
(sensitivitycategory s0 (all))
(sensitivitycategory unclassified (range documents c2))
```

level

Declare a `level` identifier in the current namespace and associate it to a previously declared `sensitivity` and zero or more categories. Note that if categories are required, then before this statement can be resolved the `sensitivitycategory` statement must be used to associate categories with the sensitivity.

Statement definition:

```
(level level_id (sensitivity_id [categoryset_id]))
```

Where:

level

The level keyword.

level_id

The level identifier.

sensitivity_id

A single previously declared sensitivity or sensitivityalias identifier.

categoryset_id

A single previously declared categoryset (named or anonymous), or a list of category and/or categoryalias identifiers. The examples show each variation.

Examples:

These `level` examples use a selection of `category`, `categoryalias` and `categoryset`'s:

```
(level systemLow (s0))
(level level_1 (s0))
(level level_2 (s0 (catrange_1)))
(level level_3 (s0 (all_cats)))
(level level_4 (unclassified (c2 c3 c4)))
```

levelrange

Declare a level range identifier in the current namespace and associate a current and clearance level.

Statement definition:

```
(levelrange levelrange_id (low_level_id high_level_id))
```

Where:

levelrange

The levelrange keyword.

levelrange_id

The levelrange identifier.

low_level_id

The current level specified by a previously declared level identifier. This may be formed by named or anonymous components as discussed in the level section and shown in the examples.

high_level_id

The clearance or high level specified by a previously declared level identifier. This may be formed by named or anonymous components as discussed in the level section and shown in the examples.

Examples:

This example policy shows `levelrange` statement and all the other MLS labeling statements discussed in this section and will compile as a standalone policy:

```
(handleunknown allow)
(mls true)

; There must be least one set of SID statements in a policy:
(sid kernel)
(sidorder (kernel))
(sidcontext kernel unconfined.context_1)
```

```
(sensitivitycategory s0 (c4 c2 c3 c1 c0 c3))

(category c0)
(categoryalias documents)
(categoryaliasactual documents c0)
(category c1)
(category c2)
(category c3)
(category c4)
(categoryalias spreadsheets)
(categoryaliasactual spreadsheets c4)

(categoryorder (c0 c1 c2 c3 spreadsheets))

(categoryset catrange_1 (range c2 c3))
(categoryset all_cats (range c0 c4))
(categoryset all_cats1 (all))

(categoryset catset_1 (documents c1))
(categoryset catset_2 (c2 c3))
(categoryset catset_3 (c4))

(categoryset just_c0 (xor (c1 c2) (documents c1 c2)))

(sensitivity s0)
(sensitivityalias unclassified)
(sensitivityaliasactual unclassified s0)

(sensitivityorder (s0))
(sensitivitycategory s0 (c0))

(sensitivitycategory s0 catrange_1)
(sensitivitycategory s0 catset_1)
(sensitivitycategory s0 catset_3)
(sensitivitycategory s0 (all))
(sensitivitycategory s0 (range documents c2))

(level systemLow (s0))
(level level_1 (s0))
(level level_2 (s0 (catrange_1)))
(level level_3 (s0 (all_cats)))
(level level_4 (unclassified (c2 c3 c4)))

(levelrange levelrange_2 (level_2 level_2))
(levelrange levelrange_1 ((s0) level_2))
```

```
(levelrange low_low (systemLow systemLow))

(context context_2 (unconfined.user object_r unconfined.object (level_1 level_3)))

; Define object_r role. This must be assigned in CIL.
(role object_r)

(block unconfined
    (user user)
    (role role)
    (type process)
    (type object)
    (userrange user (systemLow systemLow))
    (userlevel user systemLow)
    (userrole user role)
    (userrole user object_r)
    (roletype role process)
    (roletype role object)
    (roletype object_r object)

    (class file (open execute read write))

    ; There must be least one allow rule in a policy:
    (allow process self (file (read)))

    (context context_1 (user object_r object low_low))
) ; End unconfined namespace
```

rangetransition

Allows an objects level to transition to a different level. Generally used to ensure processes run with their correct MLS range, for example `init` would run at `SystemHigh` and needs to initialise / run other processes at their correct MLS range.

Statement definition:

```
(rangetransition source_id target_id class_id new_range_id)
```

Where:

rangetransition

The rangetransition keyword.

source_type_id

A single previously declared type, typealias or typeattribute identifier.

target_type_id

A single previously declared type, typealias or typeattribute identifier.

class_id

A single previously declared class or classmap identifier.

new_range_id

The new MLS range for the object class that is a previously declared levelrange identifier. This entry may also be defined as an anonymous or named level, sensitivity, sensitivityalias, category, categoryalias or categoryset identifier.

Examples:

This rule will transition the range of `sshd.exec` to `s0 - s1:c0.c3` on execution from the `init.process`:

```
(sensitivity s0)
(sensitivity s1)
(sensitivityorder (s0 s1))
(category c0)
...
(level systemlow (s0))
(level systemhigh (s1 (c0 c1 c2)))
(levelrange low_high (systemlow systemhigh))

(rangetransition init.process sshd.exec process low_high)
```

Network Labeling Statements

ipaddr

Declares a named IP address in IPv4 or IPv6 format that may be referenced by other CIL statements (i.e. `netifcon`).

Notes:

- CIL statements utilising an IP address may reference a named IP address or use an anonymous address, the examples will show each option.
- IP Addresses may be declared without a previous declaration by either writing them directly e.g. `127.0.0.11 or`::1`or by enclosing within parentheses e.g.`(127.0.0.1)`or`(::1)‘.

Statement definition:

```
(ipaddr ipaddr_id ip_address)
```

Where:

ipaddr

The ipaddr keyword.

ipaddr_id

The IP address identifier.

ip_address

A correctly formatted IP address in IPv4 or IPv6 format.

Example:

This example declares a named IP address and also passes an 'explicit anonymously declared' IP address to a macro:

```
(ipaddr netmask_1 255.255.255.0)
(context netlabel_1 (system.user object_r unconfined.object low_low))

(call build_nodecon ((192.168.1.64) netmask_1))

(macro build_nodecon ((ipaddr ARG1) (ipaddr ARG2))
    (nodecon ARG1 ARG2  netlabel_1))
```

netifcon

Label network interface objects (e.g. `eth0`).

Statement definition:

```
(netifcon netif_name netif_context_id packet_context_id)
```

Where:

netifcon

The netifcon keyword.

netif_name

The network interface name (e.g. wlan0).

netif_context_id

The security context to be allocated to the network interface.

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

packet_context_id

The security context to be allocated to packets. Note that these are defined but currently unused as the iptables(8) SECMARK services should be used to label packets.

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

These examples show named and anonymous `netifcon` statements:

```
(context context_1 (unconfined.user object_r unconfined.object low_low))
(context context_2 (unconfined.user object_r unconfined.object (systemlow level_2)))

(netifcon eth0 context_1 (unconfined.user object_r unconfined.object levelrange_1))
(netifcon eth1 context_1 (unconfined.user object_r unconfined.object ((s0) level_1)))
(netifcon eth3 context_1 context_2)
```

nodecon

Label network address objects that represent IPv4 or IPv6 IP addresses and network masks.

IP Addresses may be declared without a previous declaration by either writing them directly e.g. `127.0.0.11 or`::1`or by enclosing within parentheses e.g.`(127.0.0.1)`or`(::1)‘.

Statement definition:

```
(nodecon subnet_id netmask_id context_id)
```

Where:

nodecon

The nodecon keyword.

subnet_id

A previously declared ipaddr identifier, or an anonymous IPv4 or IPv6 formatted address.

netmask_id

A previously declared ipaddr identifier, or an anonymous IPv4 or IPv6 formatted address.

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

These examples show named and anonymous `nodecon` statements:

```
(context context_1 (unconfined.user object_r unconfined.object low_low))
(context context_2 (unconfined.user object_r unconfined.object (systemlow level_2)))

(ipaddr netmask_1 255.255.255.255)
(ipaddr ipv4_1 192.0.2.64)

(nodecon ipv4_1 netmask_1 context_2)
(nodecon 192.0.2.64 255.255.255.255 context_1)
(nodecon (192.0.2.64) netmask_1 (unconfined.user object_r unconfined.object ((s0) (s0 (

(context context_3 (sys.id sys.role my48prefix.node ((s0)(s0))))

(ipaddr netmask_2 ffff:ffff:ffff:0:0:0:0:0)
(ipaddr ipv6_2  2001:db8:1:0:0:0:0:0)

(nodecon ipv6_2 netmask_2 context_3)
(nodecon (2001:db8:1:0:0:0:0:0) (ffff:ffff:ffff:0:0:0:0:0) context_3)
(nodecon (2001:db8:1:0:0:0:0:0) netmask_2 (sys.id sys.role my48prefix.node ((s0)(s0))))
```

portcon

Label a udp, tcp, dccp or sctp port.

Statement definition:

```
(portcon protocol port|(port_low port_high) context_id)
```

Where:

portcon

The portcon keyword.

protocol

The protocol keyword tcp, udp, dccp or sctp.

port |

(port_low port_high)

A single port to apply the context, or a range of ports.

The entries must consist of numerics [0-9].

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

These examples show named and anonymous `portcon` statements:

```
(portcon tcp 1111 (unconfined.user object_r unconfined.object ((s0) (s0 (c0)))))
(portcon tcp 2222 (unconfined.user object_r unconfined.object levelrange_2))
(portcon tcp 3333 (unconfined.user object_r unconfined.object levelrange_1))
(portcon udp 4444 (unconfined.user object_r unconfined.object ((s0) level_2)))
(portcon tcp (2000 20000) (unconfined.user object_r unconfined.object (systemlow level_3
(portcon dccp (6840 6880) (unconfined.user object_r unconfined.object ((s0) level_2)))
(portcon sctp (1024 1035) (unconfined.user object_r unconfined.object ((s0) level_2)))
```

Policy Configuration Statements

mls

Defines whether the policy is built as an MLS or non-MLS policy by the CIL compiler. There MUST only be one `mls` entry in the policy otherwise the compiler will exit with an error.

Note that this can be over-ridden by the CIL compiler command line parameter `-M true|false` or `--mls true|false` flags.

Statement definition:

```
(mls boolean)
```

Where:

mls

The mls keyword.

boolean

Set to either true or false.

Example:

```
(mls true)
```

handleunknown

Defines how the kernel will handle unknown object classes and permissions when loading the policy. There MUST only be one `handleunknown` entry in the policy otherwise the compiler will exit with an error.

Note that this can be over-ridden by the CIL compiler command line parameter `-U` or `--handle-unknown` flags.

Statement definition:

```
(handleunknown action)
```

Where:

handleunknown

The handleunknown keyword.

action

A keyword of either allow, deny or reject. The kernel will handle these keywords as follows:

allow unknown class / permissions. This will set the returned AV with all 1's.

deny unknown class / permissions (the default). This will set the returned AV with all 0's.

reject loading the policy if it does not contain all the object classes / permissions.

Example:

This will allow unknown classes / permissions to be present in the policy:

```
(handleunknown allow)
```

policycap

Allow policy capabilities to be enabled via policy. These should be declared in the global namespace and be valid policy capabilities as they are checked against those known in libsepol by the CIL compiler.

Statement definition:

```
(policycap policycap_id)
```

Where:

policycap

The policycap keyword.

policycap_id

The policycap identifier (e.g. open_perms).

Example:

These set two valid policy capabilities:

```
; Enable networking controls.
(policycap network_peer_controls)

; Enable open permission check.
(policycap open_perms)
```

Role Statements

role

Declares a role identifier in the current namespace.

Statement definition:

```
(role role_id)
```

Where:

role

The role keyword.

role_id

The role identifier.

Example:

This example declares two roles: `object_r` in the global namespace and `unconfined.role`:

```
(role object_r)

(block unconfined
    (role role)
)
```

roletype

Authorises a `role` to access a `type` identifier.

Statement definition:

```
(roletype role_id type_id)
```

Where:

roletype

The roletype keyword.

role_id

A single previously declared role or roleattribute identifier.

type_id

A single previously declared type, typealias or typeattribute identifier.

Example:

This example will declare `role` and `type` identifiers, then associate them:

```
(block unconfined
    (role role)
    (type process)
    (roletype role process)
)
```

roleattribute

Declares a role attribute identifier in the current namespace. The identifier may have zero or more `role` and `roleattribute` identifiers associated to it via the `roleattributeset` statement.

Statement definition:

```
(roleattribute roleattribute_id)
```

Where:

roleattribute

The roleattribute keyword.

roleattribute_id

The roleattribute identifier.

Example:

This example will declare a role attribute `roles.role_holder` that will have an empty set:

```
(block roles
    (roleattribute role_holder)
)
```

roleattributeset

Allows the association of one or more previously declared `role` identifiers to a `roleattribute` identifier. Expressions may be used to refine the associations as shown in the examples.

Statement definition:

```
(roleattributeset roleattribute_id (role_id ... | expr ...))
```

Where:

roleattributeset

The roleattributeset keyword.

roleattribute_id

A single previously declared roleattribute identifier.

role_id

Zero or more previously declared role or roleattribute identifiers.

Note that there must be at least one role_id or expr parameter declared.

expr

Zero or more expr's, the valid operators and syntax are:

(and (role_id ...) (role_id ...))

(or (role_id ...) (role_id ...))

(xor (role_id ...) (role_id ...))

(not (role_id ...))

(all)

Example:

This example will declare three roles and two role attributes, then associate all the roles to them as shown:

```
(block roles
    (role role_1)
    (role role_2)
    (role role_3)

    (roleattribute role_holder)
    (roleattributeset role_holder (role_1 role_2 role_3))

    (roleattribute role_holder_all)
    (roleattributeset role_holder_all (all))
)
```

roleallow

Authorise the current role to assume a new role.

Notes:

- May require a `roletransition` rule to ensure transition to the new role.
- This rule is not allowed in `booleanif` statements.

Statement definition:

```
(roleallow current_role_id new_role_id)
```

Where:

roleallow

The roleallow keyword.

current_role_id

A single previously declared role or roleattribute identifier.

new_role_id

A single previously declared role or roleattribute identifier.

Example:

See the `roletransition` statement for an example.

roletransition

Specify a role transition from the current role to a new role when computing a context for the target type. The `class` identifier would normally be `process`, however for kernel versions 2.6.39 with policy version >= 25 and above, any valid class may be used. Note that a `roleallow` rule must be used to authorise the transition.

Statement definition:

```
(roletransition current_role_id target_type_id class_id new_role_id)
```

Where:

roletransition

The roletransition keyword.

current_role_id

A single previously declared role or roleattribute identifier.

target_type_id

A single previously declared type, typealias or typeattribute identifier.

class_id

A single previously declared class or classmap identifier.

new_role_id

A single previously declared role identifier to be set on transition.

Example:

This example will authorise the `unconfined.role` to assume the `msg_filter.role` role, and then transition to that role:

```
(block ext_gateway
    (type process)
    (type exec)

    (roletype msg_filter.role process)
    (roleallow unconfined.role msg_filter.role)
    (roletransition unconfined.role exec process msg_filter.role)
)
```

rolebounds

Defines a hierarchical relationship between roles where the child role cannot have more privileges than the parent.

Notes:

- It is not possible to bind the parent role to more than one child role.
- While this is added to the binary policy, it is not enforced by the SELinux kernel services.

Statement definition:

```
(rolebounds parent_role_id child_role_id)
```

Where:

rolebounds

The rolebounds keyword.

parent_role_id

A single previously declared role identifier.

child_role_id

A single previously declared role identifier.

Example:

In this example the role `test` cannot have greater privileges than `unconfined.role`:

```
(role test)

(block unconfined
    (role role)
    (rolebounds role .test)
)
```

SID Statements

sid

Declares a new SID identifier in the current namespace.

Statement definition:

```
(sid sid_id)
```

Where:

sid

The sid keyword.

sid_id

The sid identifier.

Examples:

These examples show three `sid` declarations:

```
(sid kernel)
(sid security)
(sid igmp_packet)
```

sidorder

Defines the order of sid's. This is a mandatory statement when SIDs are defined. Multiple `sidorder` statements declared in the policy will form an ordered list.

Statement definition:

```
(sidorder (sid_id ...))
```

Where:

sidorder

The sidorder keyword.

sid_id

One or more sid identifiers.

Example:

This will produce an ordered list of "`kernel security unlabeled`"

```
(sid kernel)
(sid security)
(sid unlabeled)
(sidorder (kernel security))
(sidorder (security unlabeled))
```

sidcontext

Associates an SELinux security context to a previously declared `sid` identifier.

Statement definition:

```
(sidcontext sid_id context_id)
```

Where:

sidcontext

The sidcontext keyword.

sid_id

A single previously declared sid identifier.

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Examples:

This shows two named security context examples plus an anonymous context:

```
; Two named context:
(sid kernel)
(context kernel_context (u r process low_low))
(sidcontext kernel kernel_context)

(sid security)
(context security_context (u object_r process low_low))
(sidcontext security security_context)

; An anonymous context:
(sid unlabeled)
(sidcontext unlabeled (u object_r ((s0) (s0))))
```

Type Statements

type

Declares a type identifier in the current namespace.

Statement definition:

```
(type type_id)
```

Where:

type

The type keyword.

type_id

The type identifier.

Example:

This example declares a type identifier `bluetooth.process`:

```
(block bluetooth
    (type process)
)
```

typealias

Declares a type alias in the current namespace.

Statement definition:

```
(typealias typealias_id)
```

Where:

typealias

The typealias keyword.

typealias_id

The typealias identifier.

Example:

See the `typealiasactual` statement for an example that associates the `typealias` identifier.

typealiasactual

Associates a previously declared `typealias` identifier to a previously declared `type` identifier.

Statement definition:

```
(typealiasactual typealias_id type_id)
```

Where:

typealiasactual

The typealiasactual keyword.

typealias_id

A single previously declared typealias identifier.

type_id

A single previously declared type identifier.

Example:

This example will alias `unconfined.process` as `unconfined_t` in the global namespace:

```
(typealias unconfined_t)
(typealiasactual unconfined_t unconfined.process)

(block unconfined
    (type process)
)
```

typeattribute

Declares a type attribute identifier in the current namespace. The identifier may have zero or more `type`, `typealias` and `typeattribute` identifiers associated to it via the `typeattributeset` statement.

Statement definition:

```
(typeattribute typeattribute_id)
```

Where:

typeattribute

The typeattribute keyword.

typeattribute_id

The typeattribute identifier.

Example:

This example declares a type attribute `domain` in global namespace that will have an empty set:

```
(typeattribute domain)
```

typeattributeset

Allows the association of one or more previously declared `type`, `typealias` or `typeattribute` identifiers to a `typeattribute` identifier. Expressions may be used to refine the associations as shown in the examples.

Statement definition:

```
(typeattributeset typeattribute_id (type_id ... | expr ...))
```

Where:

typeattributeset

The typeattributeset keyword.

typeattribute_id

A single previously declared typeattribute identifier.

type_id

Zero or more previously declared type, typealias or typeattribute identifiers.

Note that there must be at least one type_id or expr parameter declared.

expr

Zero or more expr's, the valid operators and syntax are:

(and (type_id ...) (type_id ...))

(or (type_id ...) (type_id ...))

(xor (type_id ...) (type_id ...))

(not (type_id ...))

(all)

Examples:

This example will take all the policy types and exclude those in `appdomain`. It is equivalent to `~appdomain` in the kernel policy language.

```
(typeattribute not_in_appdomain)

(typeattributeset not_in_appdomain (not (appdomain)))
```

This example is equivalent to `{ domain -kernel.process -ueventd.process -init.process }` in the kernel policy language:

```
(typeattribute na_kernel_or_ueventd_or_init_in_domain)

(typeattributeset na_kernel_or_ueventd_or_init_in_domain
    (and
        (and
            (and
                (domain)
                (not (kernel.process))
            )
            (not (ueventd.process))
        )
        (not (init.process))
    )
)
```

expandtypeattribute

Overrides the compiler defaults for the expansion of one or more previously declared `typeattribute` identifiers.

This rule gives more control over type attribute expansion and removal. When the value is true, all rules involving the type attribute will be expanded and the type attribute will be removed from the policy. When the value is false, the type attribute will not be removed from the policy, even if the default expand rules or "-X" option cause the rules involving the type attribute to be expanded.

Statement definition:

```
(expandtypeattribute typeattribute_id expand_value)
```

Where:

expandtypeattribute

The expandtypeattribute keyword.

typeattribute_id

One or more previously declared typeattribute identifiers. Multiple entries consist of a space separated list enclosed in parentheses '()'.

expand_value

Either true or false.

Examples:

This example uses the expandtypeattribute statement to forcibly expand a previously declared `domain` type attribute.

```
(expandtypeattribute domain true)
```

This example uses the expandtypeattribute statement to not expand previously declared `file_type` and `port_type` type attributes regardless of compiler defaults.

```
(expandtypeattribute (file_type port_type) false)
```

typebounds

This defines a hierarchical relationship between domains where the bounded domain cannot have more permissions than its bounding domain (the parent).

Requires kernel 2.6.28 and above to control the security context associated to threads in multi-threaded applications. Note that an `allow` rule must be used to authorise the bounding.

Statement definition:

```
(typebounds parent_type_id child_type_id)
```

Where:

typebounds

The typebounds keyword.

parent_type_id

A single previously declared type or typealias identifier that is the parent domain.

child_type_id

A single previously declared type or typealias identifier that is the bound (child) domain.

Example:

In this example the `httpd.child.process` cannot have `file (write)` due to lack of permissions on `httpd.process` which is the parent. It means the child domain will always have equal or less privileges than the parent:

```
(class file (getattr read write))

(block httpd
    (type process)
    (type object)
```

```
        (typebounds process child.process)
        ; The parent is allowed file 'getattr' and 'read':
        (allow process object (file (getattr read)))

        (block child
            (type process)
            (type object)

            ; However the child process has been given 'write' access that will be denied.
            (allow process httpd.object (file (read write)))
        )
    )
```

typechange

The type change rule is used to define a different label of an object for userspace SELinux-aware applications. These applications would use **security_compute_relabel(3)** and **typechange** rules in the policy to determine the new context to be applied. Note that an **allow** rule must be used to authorise the change.

Statement definition:

```
(typechange source_type_id target_type_id class_id change_type_id)
```

Where:

typechange

The typechange keyword.

source_type_id

A single previously declared type, typealias or typeattribute identifier.

target_type_id

A single previously declared type, typealias or typeattribute identifier.

class_id

A single previously declared class or classmap identifier.

change_type_id

A single previously declared type or typealias identifier that will become the new type.

Example:

Whenever **security_compute_relabel**(3) is called with the following parameters:

`scon=unconfined.object tcon=unconfined.object class=file`

the function will return a context of:

`unconfined.object:object_r:unconfined.change_label:s0`

```
(class file (getattr read write))

(block unconfined
    (type process)
    (type object)
    (type change_label)

    (typechange object object file change_label)
)
```

typemember

The type member rule is used to define a new polyinstantiated label of an object for SELinux-aware applications. These applications would use **avc_compute_member**(3) or **security_compute_member**(3) with the `typemember` rules in the policy to determine the context to be applied. The application would then manage any required polyinstantiation. Note that an `allow` rule must be used to authorise the membership.

Statement definition:

```
(typemember source_type_id target_type_id class_id member_type_id)
```

Where:

typemember

The typemember keyword.

source_type_id

A single previously declared type, typealias or typeattribute identifier.

target_type_id

A single previously declared type, typealias or typeattribute identifier.

class_id

A single previously declared class or classmap identifier.

member_type_id

A single previously declared type or typealias identifier that will become the new member type.

Example:

Whenever **avc_compute_member**(3) or **security_compute_member**(3) is called with the following parameters:

```
scon=unconfined.object tcon=unconfined.object class=file
```

the function will return a context of:

```
unconfined.object:object_r:unconfined.member_label:s0
```

```
    (class file (getattr read write))

    (block unconfined
        (type process)
        (type object)
        (type change_label)

        (typemember object object file member_label)
    )
```

typetransition

The type transition rule specifies the labeling and object creation allowed between the source_type and target_type when a domain transition is requested. Kernels from 2.6.39 with policy versions from 25 and above also support a 'name transition' rule, however this is not allowed inside conditionals and currently only supports the file classes. Note that an allow rule must be used to authorise the transition.

Statement definition:

```
    (typetransition source_type_id target_type_id class_id [object_name] default_type_id)
```

Where:

typetransition

The typetransition keyword.

source_type_id

A single previously declared type, typealias or typeattribute identifier.

target_type_id

A single previously declared type, typealias or typeattribute identifier.

class_id

A single previously declared class or classmap identifier.

object_name

A optional string within double quotes representing an object name for the 'name transition' rule. This string will be matched against the objects name (if a path then the last component of that path). If the string matches exactly, the default_type_id will then become the new type.

default_type_id

A single previously declared type or typealias identifier that will become the new type.

Examples:

This example shows a process transition rule with its supporting `allow` rule:

```
(macro domain_auto_trans ((type ARG1) (type ARG2) (type ARG3))
    ; Allow the necessary permissions.
    (call domain_trans (ARG1 ARG2 ARG3))
    ; Make the transition occur by default.
    (typetransition ARG1 ARG2 process ARG3)
)
```

This example shows a file object transition rule with its supporting `allow` rule:

```
(macro tmpfs_domain ((type ARG1))
    (type tmpfs)
    (typeattributeset file_type (tmpfs))
    (typetransition ARG1 file.tmpfs file tmpfs)
    (allow ARG1 tmpfs (file (read write execute execmod)))
)
```

This example shows the 'name transition' rule with its supporting `allow` rule:

```
(macro write_klog ((type ARG1))
    (typetransition ARG1 device.device chr_file "__kmsg__" device.klog_device)
    (allow ARG1 device.klog_device (chr_file (create open write unlink)))
    (allow ARG1 device.device (dir (write add_name remove_name)))
)
```

typepermissive

Policy database version 23 introduced the permissive statement to allow the named domain to run in permissive mode instead of running all SELinux domains in permissive mode (that was the only option prior to version 23). Note that the permissive statement only tests the source context for any policy denial.

Statement definition:

```
(typepermissive source_type_id)
```

Where:

typepermissive

The typepermissive keyword.

source_type_id

A single previously declared type or typealias identifier.

Example:

This example will allow SELinux to run the `healthd.process` domain in permissive mode even when enforcing is enabled:

```
(block healthd
    (type process)
    (typepermissive process)

    (allow ...)
)
```

typeneveraudit

Policy database version 35 introduced the neveraudit statement to suppress all AVC auditing on the named domain. This rules silences both permission denied and permission granted AVC audit messages irrespective of any auditallow, auditdeny, or dontaudit AV rules.

Statement definition:

```
(typeneveraudit source_type_id)
```

Where:

typeneveraudit

The typeneveraudit keyword.

source_type_id

A single previously declared type or typealias identifier.

Example:

This example will silence SELinux AVC audit messages for the `unconfined_t` domain:

```
(typeneveraudit unconfined_t)
```

User Statements

user

Declares an SELinux user identifier in the current namespace.

Statement definition:

```
(user user_id)
```

Where:

user

The user keyword.

user_id

The SELinux user identifier.

Example:

This will declare an SELinux user as `unconfined.user`:

```
(block unconfined
    (user user)
)
```

userrole

Associates a previously declared `user` identifier with a previously declared `role` identifier.

Statement definition:

```
(userrole user_id role_id)
```

Where:

userrole

The userrole keyword.

user_id

A previously declared SELinux user or userattribute identifier.

role_id

A previously declared role or roleattribute identifier.

Example:

This example will associate `unconfined.user` to `unconfined.role`:

```
(block unconfined
    (user user)
    (role role)
    (userrole user role)
)
```

userattribute

Declares a user attribute identifier in the current namespace. The identifier may have zero or more `user` and `userattribute` identifiers associated to it via the `userattributeset` statement.

Statement definition:

```
(userattribute userattribute_id)
```

Where:

userattribute

The userattribute keyword.

userattribute_id

The userattribute identifier.

Example:

This example will declare a user attribute `users.user_holder` that will have an empty set:

```
(block users
    (userattribute user_holder)
)
```

userattributeset

Allows the association of one or more previously declared `user` or `userattribute` identifiers to a `userattribute` identifier. Expressions may be used to refine the associations as shown in the examples.

Statement definition:

```
(userattributeset userattribute_id (user_id ... | expr ...))
```

Where:

userattributeset

The userattributeset keyword.

userattribute_id

A single previously declared userattribute identifier.

user_id

Zero or more previously declared user or userattribute identifiers.

Note that there must be at least one user_id or expr parameter declared.

expr

Zero or more expr's, the valid operators and syntax are:

(and (user_id ...) (user_id ...))

(or (user_id ...) (user_id ...))

(xor (user_id ...) (user_id ...))

(not (user_id ...))

(all)

Example:

This example will declare three users and two user attributes, then associate all the users to them as shown:

```
(block users
    (user user_1)
    (user user_2)
    (user user_3)

    (userattribute user_holder)
    (userattributeset user_holder (user_1 user_2 user_3))

    (userattribute user_holder_all)
    (userattributeset user_holder_all (all))
)
```

userlevel

Associates a previously declared `user` identifier with a previously declared `level` identifier. The `level` may be named or anonymous.

Statement definition:

```
(userlevel user_id level_id)
```

Where:

userlevel

The userlevel keyword.

user_id

A previously declared SELinux user identifier.

level_id

A previously declared level identifier. This may consist of a single sensitivity with zero or more mixed named and anonymous category's as discussed in the level statement.

Example:

This example will associate `unconfined.user` with a named `level` of `systemlow`:

```
(sensitivity s0)
(level systemlow (s0))

(block unconfined
    (user user)
     (userlevel user systemlow)
    ; An anonymous example:
    ;(userlevel user (s0))
)
```

userrange

Associates a previously declared `user` identifier with a previously declared `levelrange` identifier. The `levelrange` may be named or anonymous.

Statement definition:

```
(userrange user_id levelrange_id)
```

Where:

userrange

The userrange keyword.

user_id

A previously declared SELinux user identifier.

levelrange_id

A previously declared levelrange identifier. This may be formed by named or anonymous components as discussed in the levelrange statement and shown in the examples.

Example:

This example will associate `unconfined.user` with a named `levelrange` of `low_high`, other anonymous examples are also shown:

```
(category c0)
(category c1)
(categoryorder (c0 c1))
(sensitivity s0)
(sensitivity s1)
(sensitivityorder (s0 s1))
(sensitivitycategory s0 (c0 c1))
(level systemLow (s0))
(level systemHigh (s0 (c0 c1)))
(levelrange low_high (systemLow systemHigh))

(block unconfined
    (user user)
    (role role)
    (userrole user role)
    ; Named example:
    (userrange user low_high)
    ; Anonymous examples:
    ;(userrange user (systemLow systemHigh))
    ;(userrange user (systemLow (s0 (c0 c1))))
    ;(userrange user ((s0) (s0 (c0 c1))))
)
```

userbounds

Defines a hierarchical relationship between users where the child user cannot have more privileges than the parent.

Notes:

- It is not possible to bind the parent to more than one child.
- While this is added to the binary policy, it is not enforced by the SELinux kernel services.

Statement definition:

```
(userbounds parent_user_id child_user_id)
```

Where:

userbounds

The userbounds keyword.

parent_user_id

A previously declared SELinux user identifier.

child_user_id

A previously declared SELinux user identifier.

Example:

The user `test` cannot have greater privileges than `unconfined.user`:

```
(user test)

(unconfined
    (user user)
    (userbounds user .test)
)
```

userprefix

Declare a user prefix that will be replaced by the file labeling utilities described at https://github.com/SELinuxProject/selinux-notebook/blob/main/src/policy_store_config_files.md that details the `file_contexts` entries.

Statement definition:

```
(userprefix user_id prefix)
```

Where:

userprefix

The userprefix keyword.

user_id

A previously declared SELinux user identifier.

prefix

The string to be used by the file labeling utilities.

Example:

This example will associate `unconfined.admin` user with a prefix of "`user`":

```
(block unconfined
    (user admin)
    (userprefix admin user)
)
```

selinuxuser

Associates a GNU/Linux user to a previously declared `user` identifier with a previously declared MLS `userrange`. Note that the `userrange` is required even if the policy is non-MCS/MLS.

Statement definition:

```
(selinuxuser user_name user_id userrange_id)
```

Where:

selinuxuser

The selinuxuser keyword.

user_name

A string representing the GNU/Linux user name

user_id

A previously declared SELinux user identifier.

userrange_id

A previously declared userrange identifier that has been associated to the user identifier. This may be formed by named or anonymous components as discussed in the userrange statement and shown in the examples.

Example:

This example will associate `unconfined.admin` user with a GNU / Linux user "`admin_1`":

```
(block unconfined
    (user admin)
    (selinuxuser admin_1 admin low_low)
)
```

selinuxuserdefault

Declares the default SELinux user. Only one `selinuxuserdefault` statement is allowed in the policy. Note that the `userrange` identifier is required even if the policy is non-MCS/MLS.

Statement definition:

```
(selinuxuserdefault user_id userrange_id)
```

Where:

selinuxuserdefault

The selinuxuserdefault keyword.

user_id

A previously declared SELinux user identifier.

userrange_id

A previously declared userrange identifier that has been associated to the user identifier. This may be formed by named or anonymous components as discussed in the userrange statement and shown in the examples.

Example:

This example will define the `unconfined.user` as the default SELinux user:

```
(block unconfined
    (user user)
    (selinuxuserdefault user low_low)
)
```

Infiniband Statements

To support access control for InfiniBand (IB) partitions and subnet management, security contexts are provided for: Partition Keys (Pkey) that are 16 bit numbers assigned to subnets and their IB end ports. An overview of the SELinux IB implementation can be found at: http://marc.info/?l=selinux&m=149519833917911&w=2.

ibpkeycon

Label IB partition keys. This may be a single key or a range.

Statement definition:

```
(ibpkeycon subnet pkey|(pkey_low pkey_high)  context_id)
```

Where:

ibpkeycon

The ibpkeycon keyword.

subnet

IP address in IPv6 format.

pkey | (pkey_low pkey_high)

A single partition key or a range of partition keys.

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for a partition key range of `0x0-0x10` assigned to an IPv6 subnet:

```
(ibpkeycon fe80:: (0 0x10) (system_u system_r kernel_t (low (s3 (cats01 cats02)))))
```

ibendportcon

Label IB end ports.

Statement definition:

```
(ibendportcon device_id port context_id)
```

Where:

ibendportcon

The ibendportcon keyword.

device_id

A single device identifier.

port

A single port number.

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

A named context for device `mlx5_0` on port `1`:

```
(ibendportcon mlx5_0 1 system_u_bin_t_l2h)
```

Xen Statements

Policy version 30 introduced the `devicetreecon` statement and also expanded the existing I/O memory range to 64 bits in order to support hardware with more than 44 bits of physical address space (32-bit count of 4K pages).

See the "XSM/FLASK Configuration" document for further information ()

iomemcon

Label i/o memory. This may be a single memory location or a range.

Statement definition:

```
(iomemcon mem_addr|(mem_low mem_high) context_id)
```

Where:

iomemcon

The iomemcon keyword.

mem_addr |

(mem_low mem_high)

A single memory address to apply the context, or a range of addresses.

The entries must consist of numerics [0-9].

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for a memory address range of `0xfebe0-0xfebff`:

```
(iomemcon (1043424 1043455) (unconfined.user object_r unconfined.object low_low))
```

ioportcon

Label i/o ports. This may be a single port or a range.

Statement definition:

```
(ioportcon port|(port_low port_high) context_id)
```

Where:

ioportcon

The ioportcon keyword.

port |

(port_low port_high)

A single port to apply the context, or a range of ports.

The entries must consist of numerics [0-9].

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for a single port of :`0xecc0`:

```
(ioportcon 60608 (unconfined.user object_r unconfined.object low_low))
```

pcidevicecon

Label a PCI device.

Statement definition:

```
(pcidevicecon device context_id)
```

Where:

pcidevicecon

The pcidevicecon keyword.

device

The device number.The entries must consist of numerics [0-9].

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for a pci device address of `0xc800`:

```
(pcidevicecon 51200 (unconfined.user object_r unconfined.object low_low))
```

pirqcon

Label an interrupt level.

Statement definition:

```
(pirqcon irq_level context_id)
```

Where:

pirqcon

The pirqcon keyword.

irq_level

The interrupt request number. The entries must consist of numerics [0-9].

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for IRQ 33:

```
(pirqcon 33 (unconfined.user object_r unconfined.object low_low))
```

devicetreecon

Label device tree nodes.

Statement definition:

```
(devicetreecon path context_id)
```

Where:

devicetreecon

The devicetreecon keyword.

path

The device tree path. If this contains spaces enclose within "".

context_id

A previously declared context identifier or an anonymous security context (user role type levelrange), the range MUST be defined whether the policy is MLS/MCS enabled or not.

Example:

An anonymous context for the specified path:

```
(devicetreecon "/this is/a/path" (unconfined.user object_r unconfined.object low_low))
```

Example Policy

```
(type bin_t)
(type kernel_t)
(type security_t)
(type unlabeled_t)
(handleunknown allow)
(mls true)

(policycap open_perms)

(category c0)
(category c1)
(category c2)
(category c3)
(category c4)
(category c5)
(categoryalias cat0)
```

```
(categoryaliasactual cat0 c0)
(categoryset cats01 (c0 c1))
(categoryset cats02 (c2 c3))
(categoryset cats03 (range c0 c5))
(categoryset cats04 (not (range c0 c2)))
(categoryorder (cat0 c1 c2 c3))
(categoryorder (c3 c4 c5))

(sensitivity s0)
(sensitivity s1)
(sensitivity s2)
(sensitivity s3)
(sensitivityalias sens0)
(sensitivityaliasactual sens0 s0)
(sensitivityorder (s0 s1 s2 s3))

(sensitivitycategory s0 (cats03))
(sensitivitycategory s1 cats01)
(sensitivitycategory s1 (c2))
(sensitivitycategory s2 (cats01 cats02))
(sensitivitycategory s2 (range c4 c5))
(sensitivitycategory s3 (range c0 c5))

(level low (s0))
(level high (s3 (range c0 c3)))
(levelrange low_high (low high))
(levelrange lh1 ((s0 (c0)) (s2 (c0 c3))))
(levelrange lh2 (low (s2 (c0 c3))))
(levelrange lh3 ((s0 cats04) (s2 (range c0 c5))))
(levelrange lh4 ((s0) (s1)))

(block policy
    (class file (execute_no_trans entrypoint execmod open audit_access a b c d e))
    (class socket (nlmsg))
    ; order should be: file socket char b c a dir d e f
    (classorder (file socket char))
    (classorder (unordered dir))
    (classorder (unordered c a b d e f))
    (classorder (char b c a))

    (common file (ioctl read write create getattr setattr lock relabelfrom
            relabelto append unlink link rename execute swapon
            quotaon mounton))
    (classcommon file file)

    (classpermission file_rw)
```

```
(classpermissionset file_rw (file (read write getattr setattr lock append)))

;;(classpermission loop1)
;;(classpermissionset loop1 ((loop2)))
;;(classpermission loop2)
;;(classpermissionset loop2 ((loop3)))
;;(classpermission loop3)
;;(classpermissionset loop3 ((loop1)))

(class char (foo))
(classcommon char file)

(class dir ())
(class a ())
(class b ())
(class c ())
(class d ())
(class e ())
(class f ())
(classcommon dir file)

(classpermission char_w)
(classpermissionset char_w (char (write setattr)))
(classpermissionset char_w (file (open read getattr)))

(classmap files (read))
(classmapping files read
    (file (open read getattr)))
(classmapping files read
    char_w)

(type auditadm_t)
(type console_t)
(type console_device_t)
(type user_tty_device_t)
(type device_t)
(type getty_t)
(type exec_t)
(type bad_t)

;;(allow console_t console_device_t file_rw)
(allow console_t console_device_t (files (read)))

(permissionx ioctl_test (ioctl files (and (range 0x1600 0x19FF) (not (range 0x1750 0x175
(allowx console_t console_device_t ioctl_test)
```

```
(boolean secure_mode false)
(boolean console_login true)

(sid kernel)
(sid security)
(sid unlabeled)
(sidorder (kernel security))
(sidorder (security unlabeled))

(typeattribute exec_type)
(typeattribute foo_type)
(typeattribute bar_type)
(typeattribute baz_type)
(typeattribute not_bad_type)
(typeattributeset exec_type (or bin_t kernel_t))
(typeattributeset foo_type (and exec_type kernel_t))
(typeattributeset bar_type (xor exec_type foo_type))
(typeattributeset baz_type (not bin_t))
(typeattributeset baz_type (and exec_type (and bar_type bin_t)))
(typeattributeset not_bad_type (not bad_t))
(typealias sbin_t)
(typealiasactual sbin_t bin_t)
(typepermissive device_t)
(typeneveraudit device_t)
(typemember device_t bin_t file exec_t)
(typemember exec_type self file exec_t)
(typetransition device_t console_t files console_device_t)
(typetransition device_t exec_type files console_device_t)
(typetransition exec_type self files console_device_t)
(typetransition exec_type self files "filename" console_device_t)
(typechange console_device_t device_t file user_tty_device_t)
(typechange exec_type device_t file user_tty_device_t)
(typechange exec_type self file console_device_t)

(roleattribute exec_role)
(roleattribute foo_role)
(roleattribute bar_role)
(roleattribute baz_role)
(roleattribute foo_role_a)
(roleattributeset exec_role (or user_r system_r))
(roleattributeset foo_role_a (baz_r user_r system_r))
(roleattributeset foo_role (and exec_role system_r))
(roleattributeset bar_role (xor exec_role foo_role))
(roleattributeset baz_role (not user_r))

(rangetransition device_t console_t file low_high)
```

```
(rangetransition device_t kernel_t file ((s0) (s3 (not c3))))

(typetransition device_t console_t file "some_file" getty_t)

(allow foo_type self (file (execute)))
(allow bin_t device_t (file (execute)))

;; Next two rules violate the neverallow rule that follows
;;(allow bad_t not_bad_type (file (execute)))
;;(allow bad_t exec_t (file (execute)))
(neverallow bad_t not_bad_type (file (execute)))

(auditallowx getty_t console_device_t (ioctl file (range 0x1000 0x10FF)))
(auditallowx getty_t kernel_t (nlmsg socket (range 0x1000 0x10FF)))

(booleanif secure_mode
    (true
        (auditallow device_t exec_t (file (read write)))
    )
)

(booleanif console_login
    (true
        (typechange auditadm_t console_device_t file user_tty_device_t)
        (allow getty_t console_device_t (file (getattr open read write append)))
        (auditallowx getty_t console_device_t (ioctl file (range 0x2000 0x21FF)))
        (auditallowx getty_t kernel_t (nlmsg socket (0x1)))
    )
    (false
        (dontaudit getty_t console_device_t (file (getattr open read write append)))
        (dontauditx getty_t console_device_t (ioctl file (range 0x3000 0x31FF)))
    )
)

(booleanif (not (xor (eq secure_mode console_login)
            (and (or secure_mode console_login) secure_mode ) ) )
    (true
        (allow bin_t exec_t (file (execute)))
    )
)

(tunable allow_execfile true)
(tunable allow_userexec false)

(tunableif (not (xor (eq allow_execfile allow_userexec)
            (and (or allow_execfile allow_userexec)
```

```
                (and allow_execfile allow_userexec) ) ) )
    (true
        (allow bin_t exec_t (file (execute)))
    )
)

(optional allow_rules
    (allow user_t exec_t (bins (execute)))
)

(dontaudit device_t auditadm_t (file (read)))
(auditallow device_t auditadm_t (file (open)))

(user system_u)
(user user_u)
(user foo_u)
(userprefix user_u user)
(userprefix system_u user)

(selinuxuser name user_u low_high)
(selinuxuserdefault user_u ((s0 (c0)) (s3 (range c0 c3))))

(role system_r)
(role user_r)
(role baz_r)

(roletype system_r bin_t)
(roletype system_r kernel_t)
(roletype system_r security_t)
(roletype system_r unlabeled_t)
(roletype system_r exec_type)
(roletype exec_role bin_t)
(roletype exec_role exec_type)
(roleallow system_r user_r)
(roletransition system_r bin_t file user_r)

(userrole foo_u foo_role)
(userlevel foo_u low)

(userattribute ua1)
(userattribute ua2)
(userattribute ua3)
(userattribute ua4)
(userattributeset ua1 (user_u system_u))
(userattributeset ua2 (foo_u system_u))
(userattributeset ua3 (and ua1 ua2))
```

```
(user u5)
(user u6)
(userlevel u5 low)
(userlevel u6 low)
(userrange u5 low_high)
(userrange u6 low_high)
(userattributeset ua4 (u5 u6))
(userrole ua4 foo_role_a)

(userrange foo_u low_high)

(userrole system_u system_r)
(userlevel system_u low)
(userrange system_u low_high)

(userrole user_u user_r)
(userlevel user_u (s0 (range c0 c2)))
(userrange user_u (low high))

(sidcontext kernel (system_u system_r kernel_t ((s0) high)))
(sidcontext security (system_u system_r security_t (low (s3 (range c0 c3)))))
(sidcontext unlabeled (system_u system_r unlabeled_t (low high)))

(context system_u_bin_t_l2h (system_u system_r bin_t (low high)))

(ipaddr ip_v4 192.25.35.200)
(ipaddr netmask 192.168.1.1)
(ipaddr ip_v6 2001:0DB8:AC10:FE01::)
(ipaddr netmask_v6 2001:0DE0:DA88:2222::)

(filecon "/usr/bin/foo" file system_u_bin_t_l2h)
(filecon "/usr/bin/bar" file (system_u system_r kernel_t (low low)))
(filecon "/usr/bin/baz" any ())
(filecon "/usr/bin/aaa" any (system_u system_r kernel_t ((s0) (s3 (range c0 c2)))))
(filecon "/usr/bin/bbb" any (system_u system_r kernel_t ((s0 (c0)) high)))
(filecon "/usr/bin/ccc" any (system_u system_r kernel_t (low (s3 (cats01)))))
(filecon "/usr/bin/ddd" any (system_u system_r kernel_t (low (s3 (cats01 cats02)))))
(nodecon ip_v4 netmask system_u_bin_t_l2h)
(nodecon ip_v6 netmask_v6 system_u_bin_t_l2h)
(portcon udp 25 system_u_bin_t_l2h)
(portcon tcp 22 system_u_bin_t_l2h)
(portcon dccp (2048 2096) system_u_bin_t_l2h)
(portcon sctp (1024 1035) system_u_bin_t_l2h)
(genfscon - "/usr/bin" system_u_bin_t_l2h)
(netifcon eth0 system_u_bin_t_l2h system_u_bin_t_l2h) ;different contexts?
(netifcon podman* system_u_bin_t_l2h system_u_bin_t_l2h)
```

```
    (fsuse xattr ext3 system_u_bin_t_l2h)

    ; XEN
    (pirqcon 256 system_u_bin_t_l2h)
    (iomemcon (0 255) system_u_bin_t_l2h)
    (ioportcon (22 22) system_u_bin_t_l2h)
    (pcidevicecon 345 system_u_bin_t_l2h)
    (devicetreecon "/this is/a/path" system_u_bin_t_l2h)

    ; InfiniBand
    (ibpkeycon fe80:: (0 0x10) system_u_bin_t_l2h)
    (ibpkeycon fe80::7629:afff:fe0f:8e5d (15 25) (system_u system_r kernel_t (low (s3 (cats(
    (ibendportcon mlx5_0 1 system_u_bin_t_l2h)
    (ibendportcon mlx4_3 5 (system_u system_r kernel_t (low (s3 (cats01 cats02)))))

    (constrain (files (read)) (not (or (and (eq t1 exec_t) (eq t2 bin_t)) (eq r1 r2))))
    (constrain char_w (not (or (and (eq t1 exec_t) (eq t2 bin_t)) (eq r1 r2))))

    (constrain (file (read)) (or (and (eq t1 exec_t) (neq t2 bin_t) ) (eq u1 ua4) ) )
    (constrain (file (open)) (dom r1 r2))
    (constrain (file (open)) (domby r1 r2))
    (constrain (file (open)) (incomp r1 r2))

    (validatetrans file (eq t1 exec_t))

    (mlsconstrain (file (open)) (not (or (and (eq l1 l2) (eq u1 u2)) (eq r1 r2))))
    (mlsconstrain (file (open)) (or (and (eq l1 l2) (eq u1 u2)) (neq r1 r2)))
    (mlsconstrain (file (open)) (dom h1 l2))
    (mlsconstrain (file (open)) (domby l1 h2))
    (mlsconstrain (file (open)) (incomp l1 l2))

    (mlsvalidatetrans file (domby l1 h2))

    (macro test_mapping ((classpermission cps))
           (allow bin_t auditadm_t cps))

    (call test_mapping ((file (read))))
    (call test_mapping ((files (read))))
    (call test_mapping (char_w))

    (defaultuser (file char) source)
    (defaultrole char target)
    (defaulttype (files) source)
    (defaultrange (file) target low)
    (defaultrange (char) source low-high)
)
```

```
(macro all ((type x))
    (allow x bin_t (policy.file (execute)))
    (allowx x bin_t (ioctl policy.file (range 0x1000 0x11FF)))
)
(call all (bin_t))

(block z
    (block ba
        (roletype r t)
        (blockabstract z.ba)))

(block test_ba
    (blockinherit z.ba)
    (role r)
    (type t))

(block bb
    (type t1)
    (type t2)
    (boolean b1 false)
    (tunable tun1 true)
    (macro m ((boolean b))
        (tunableif tun1
            (true
                (allow t1 t2 (policy.file (write))))
            (false
                (allow t1 t2 (policy.file (execute)))))
        (booleanif b
            (true
                (allow t1 t2 (policy.file (read))))))

    (call m (b1))
)

(in bb
    (tunableif bb.tun1
        (true
            (allow bb.t2 bb.t1 (policy.file (read write execute))))))
```