Policy DSL: High-level Specifications of Information Flows for Security Policies

Joe Hurd, Magnus Carlsson, Sigbjorn Finne,
Brett Letner, Joel Stanley and Peter White
Galois, Inc.

{joe,magnus,sof,bletner,jstanley,peter}@galois.com

15 April 2009

Abstract

SELinux security policies are powerful tools to implement properties such as *process confinement* and *least privilege*. They can also be used to support MLS policies on SELinux. However, the policies are very complex, and creating them is a difficult and error-prone process. Furthermore, it is not possible to state explicit constraints on an SELinux policy such as "information flowing to the network must be encrypted".

We present two related Domain Specific Languages (DSL) to make it much easier to specify SELinux security policies. The first DSL is called Lobster. Lobster allows the user to state an information flow policy at a very high level of abstraction, and then refine the policy into lower and lower levels until it can be translated by the Lobster compiler into an SELinux policy. The second DSL is called Symbion. Symbion allows the user to state policy constraints such as "information flowing to the network must be encrypted". Symbion and Lobster will then interface with tools that can check that the policy satisfies the constraints.

We also present the analysis tool Shrimp, with which we are able to analyze and find errors in the SELinux Reference Policy. Shrimp is also the basis for our ongoing effort in reverse translating the Reference Policy into Lobster.

Since Lobster and Symbion represent information flows and contraints on information flows, they are more broadly applicable than to just SELinux. We will point to some of the directions we wish to take Lobster and Symbion beyond SELinux.

Contents

1 Introduction

SELinux provides a powerful enforcement mechanism called type enforcement [7, 8] that permits much finer-grained control of access to resources than the simple DAC and user-id based mechanisms in standard Unix or Linux operating systems. The finer grained access control permits the implementation of the principle of least privilege. In particular, SELinux systems need not have an all powerful root login, which is often the target of attempts to compromise the system.

The cost of the increased granularity of access control has been a corresponding increase in the complexity of the policy specification. Current type enforcement policy specifications have tens of thousands of statements. The size of the specifications has been cut down somewhat by a slightly higher-level policy specification language based on the m4 macro preprocessor (and other Unix tools). The m4 based language is called the *reference policy* language. However the policy specifications are still large and complex. In addition, the semantics of the m4-based language is uncertain. Neither of these languages gives any assurance to the user that a higher-level policy such as "domain A is isolated from domain B" has been correctly implemented in the low level policy statement.

Thus we address two concerns with the current SELinux policy languages, their complexity and the lack of facility to make explicit properties that constrain the policy. To address these concerns, we have designed two tightly-coupled but independent domain-specific languages: Lobster, a language for expressing security policies at a high level; and Symbion, an assertion language over information flows.

Addressing the concern for complexity of the policy specification, a Lobster specification clearly expresses a policy in terms of information flows among nested security domains. Lobster also supports a notion of refinement, where an overall, concise specification can be turned into more detailed specifications in a way that ensures that the overall policy is upheld.

Addressing the concern to state explicit constraints on the policy, Symbion allows one to make assertions about the nature of information flow in a given system. For example, in Symbion we can express the restriction "every flow from the Secret domain to the Internet domain goes through the Encrypt domain."

A Lobster policy is refined a step at a time from a very high level down to a level that is very close to the current reference policy. The highest level Lobster domains might specify the desired information flows amongst a few application domains. The lowest level Lobster domains correspond closely to primitive SELinux classes such as *File* and *Process*. At each stage of refinement more details are added to the information flows in the policy.

The Symbion constraints can be applied at any level of refinement. When a domain is refined, the constraints on the domain must be observed by the refinement of the domain as well. Symbion thus imposes consistency conditions on the refinement that must be discharged as the specification is refined. Symbion is being designed to permit model checking tools to automatically check

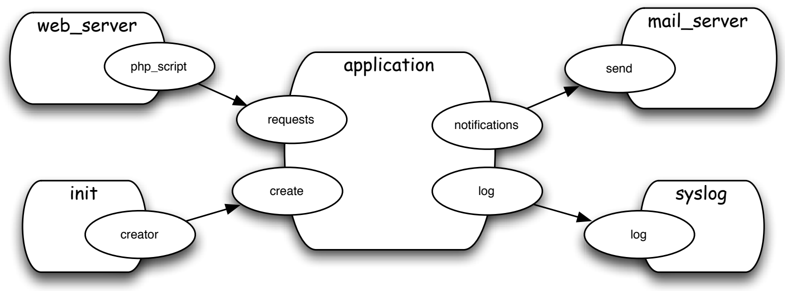


Figure 1: Modeling an application in an SELinux system.

the consistency of refinements.

In this paper, we show how to compile Lobster policies into SELinux modules, and present early tools for analyzing the complete Reference Policy and generating the corresponding Lobster specifications. The tools discover nested security domains and make all implicit information flows explicit. The resulting Lobster specifications are informing ongoing work in the design of visualization tools for large Lobster specifications.

Lobster began as a high level language specifically for the purpose of specifying SELinux policies. We have observed that the concepts of the language are more broadly applicable that just SELinux. Thus we have separated out the few pieces of the language that apply specifically to SELinux, so that the SELinux version of Lobster is an instance of a more general language. Lobster and Symbion are part of a generic approach that we hope to apply to larger systems, where SELinux policies are just one piece of the puzzle. Such large systems may consist of networks of computers running different operating systems, but also routers, firewalls and guards. The goal is to have a single language (that we call Nova Scotia) in which we can specify security policies for such large systems.

2 Modeling SELinux Systems

A security policy designer might view an SELinux system in terms of information flows between security domains. For example, the application in Figure 1 receives information from a web server and sends information to a mail server; the arrows in the diagram indicate the dominant flow of information.

Information flow can also represent privileges or capabilities, in the following way: if subject A has the privilege to act on object B, then this can be represented by an information flow between A and B. The dominant flow of a write privilege would be from A to B, and conversely the dominant flow of a read privilege would be from B to A. For another example, the `init` process in Figure 1 has the privilege to start up the application, and this is represented by an information flow from the Init domain to the Application domain.

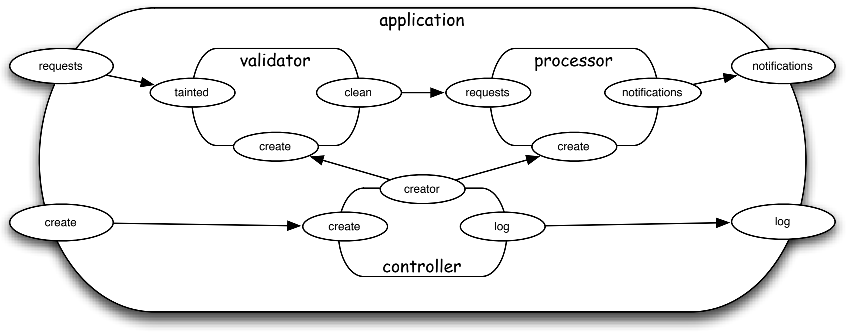


Figure 2: Inside the SELinux application.

```
allow application_t lib_t:dir { getattr search };
allow application_t ld_so_t:file { getattr read execute ioctl };
allow application_t usr_t:dir { getattr search read lock ioctl };
allow application_t syslogd_t:unix_dgram_socket sendto;
allow application_t syslogd_t:unix_stream_socket connectto;
```

Figure 3: Example SELinux native policy statements.

Representing privilege with information flow is a different way of viewing a system, and this shift of viewpoint emphasizes different aspects of the system. Viewing a system in terms of privileges makes it important to distinguish subjects and objects, but from an information flow perspective there is no difference between A having the power to write to B and B having the power to read from A.[1] Emphasizing flow makes it easier to see how information flows through a system. Returning again to Figure 1, the policy representation makes it easy to see that there is a flow of information from the web server to the mail server moderated by the application.

SELinux applications are not restricted to run within a single security domain: separate process and resources can exist in their own security domains, and the SELinux system is used to ensure that all interactions between them are consistent with the security policy in force. Figure 2 shows how the example application might be internally broken down into separate security domains, with explicit information flows permitted between them.

Continuing with the example, suppose that a developer implements the application and also writes an SELinux policy for it. Some example policy statements are shown in Figure 3. How is it possible to check that the security policy designer's permitted information flows and the developer's SELinux policy statements are consistent?

[1]In fact, the distinction between subjects and objects can be encoded in an information flow view, and in Section 5.1 such an encoding is used to compile an information flow policy to SELinux.

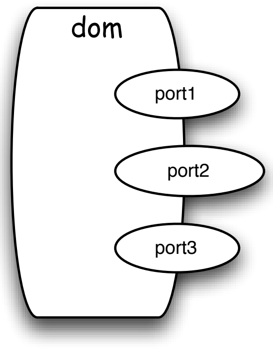


Figure 4: A Lobster domain with named ports.

One approach to solving this problem is to express the information flows in a high level language, and to compile it down to SELinux policies. This paper presents *Lobster*, a Domain Specific Language (DSL) for expressing information flows between security domains.

In general, the SELinux policy written by the developer will contain much more detail than the information flow view of the security policy designer. This is the intended use of the Lobster DSL:

1. A security policy designer writes a Lobster information flow diagram for the application.

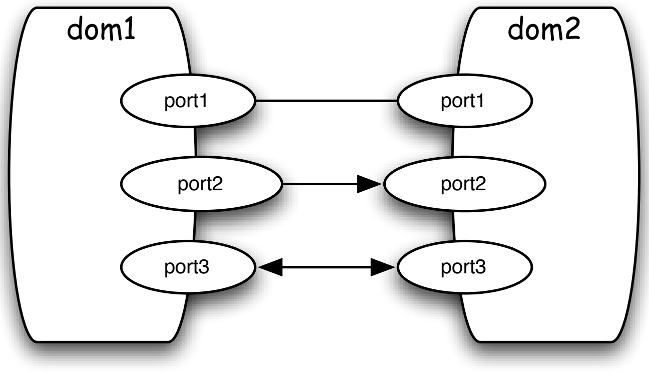
2. A developer writes a Lobster policy for the application.

3. An automatic tool verifies that the Lobster policy (2) is a refinement of the Lobster information flow diagram (1), in that *no extra information flows have been introduced*.

4. A compiler takes the Lobster policy (2) and generates SELinux policy statements.

3 Information Flow Graphs

In the previous section, Figures 1 and 2 presented information flows between nested security domains in a graphical form. Such diagrams are called *information flow graphs*, and these graphs are the semantic foundation of Lobster policies. A Lobster policy is compiled into an information flow graph as an intermediate step on the way to a native SELinux policy.



```
dom1.port1 -- dom2.port1;
dom1.port2 --> dom2.port2;
dom1.port3 <--> dom2.port3;
```

Figure 5: Example Lobster connections.

3.1 Domains

Figure 4 shows a Lobster domain with named ports. A domain is a security domain, and the inside can communicate with the outside only via its ports.

3.2 Ports

Ports can be declared with information flow properties:

```
port p1 : {direction = input, type = requests};
port p2 : {direction = bidirectional};
```

All information flow properties are optional: an unspecified property is deemed to be polymorphic in its value. When specified, the `direction` property is either `input`, `output` or `bidirectional`, and restricts the kind of connections that can be made to the port. The `type` value captures the type of information handled by the port—it has nothing to do with SELinux types.

Although `direction` and `type` are the only information flow properties provided by Lobster, the language is designed to be extensible in this regard. For example, to compile Lobster policies to SELinux native policy, we will define and use a `position` information flow property that encodes the subject/object relationship that is central to SELinux type enforcement.

3.3 Connections

Pairs of domain ports may be connected: Figure 5 shows three examples. A connection is legal between two ports if they have the same `type` properties, and the different kinds of connection direction symbols add the following requirements:

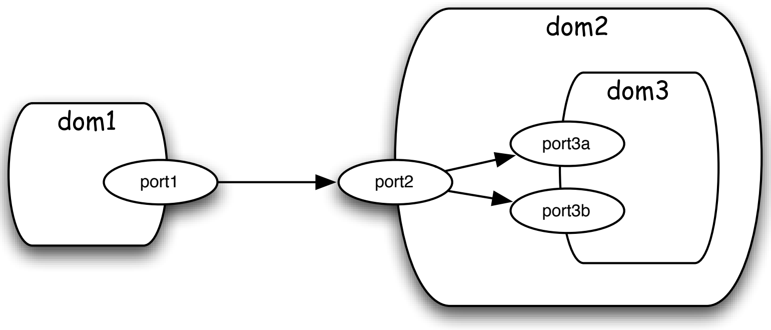


Figure 6: Lobster domains can be nested.

- `dom1.port1 -- dom2.port1;` adds no requirements;
- `dom1.port2 --> dom2.port2;` requires that `dom1.port2` has an `output direction` property, and `dom2.port2` has an `input direction` property.
- `dom1.port3 <--> dom2.port3;` requires that the `dom1.port3` has a `bidirectional direction` property, and `dom2.port3` has an `bidirectional direction` property.

As we will see in Section 5, Lobster connections are compiled into SELinux `type` and `allow` statements.

3.4 Nested Domains

One domain may be nested inside another; connections are permitted between ports in the containing domain and ports in the nested domain. Figure 6 shows an example of connections between nested domains: note that `port1` is indirectly connected via `port2` to both `port3a` and `port3b`.

A domain corresponds to an SELinux type, where the name of the type is determined by the name of the domain and the names of the containing domains. For example, the SELinux type corresponding to the domain in Figure 4 is simply `dom_t`, and the name of the *controller* domain in Figure 2 is `application_controller_t`.

3.5 Internal Connections

It is possible to connect one port of a domain to another port of the same domain; such a connection can occur on either the outside or the inside of the domain. Such a connection on the inside of the domain is called an *internal connection*; all other connections are *regular connections*. There is an important difference between internal connections and regular connections, which is that

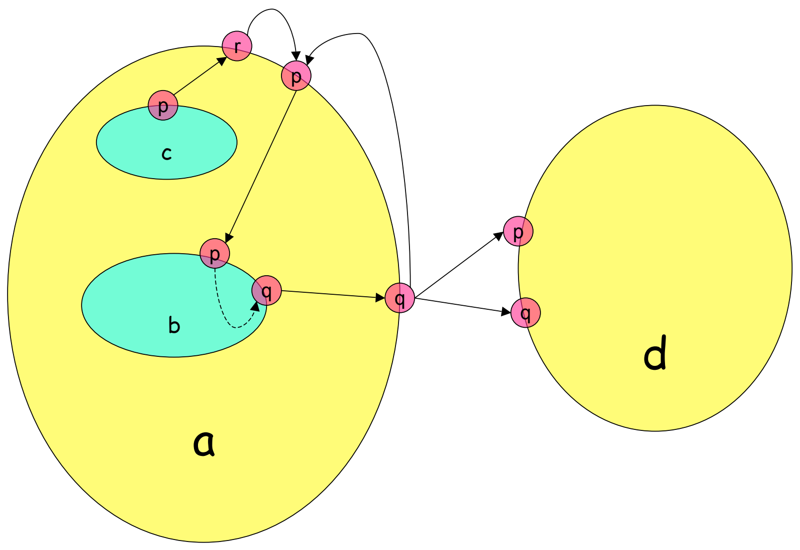


Figure 7: An example information flow graph.

internal connections are allowed to connect ports with incompatible information flow properties.

Figure 7 shows an example information flow graph, with domains a, b, c and d, each having ports named p, q or r, and solid arrows being regular connections between the ports. The regular connection from $a.r$ to $a.p$ is an example of two ports of the a domain being connected together, with the connection outside a. The dotted arrow from $b.p$ to $b.q$ is an internal connection, connecting together two ports of the b domain, with the connection being inside b.

3.6 Flows

A *flow* in G between ports $d_0.p_0$ and $d_n.p_n$ is an alternating sequence of connections and ports

$$\langle l_0 \rangle \; d_1.p_1 \; \langle l_1 \rangle \; d_2.p_2 \; \cdots \; d_{n-1}.p_{n-1} \; \langle l_{n-1} \rangle$$

where l_i is a directed connection in G from port $d_i.p_i$ to port $d_{i+1}.p_{i+1}$, and exactly one of l_i and l_{i+1} is a connection inside the domain d_{i+1}. The latter condition enforces correct flow through domain ports: inside the domain to outside, or outside to inside (but not inside to inside or outside to outside). Note that the connections in flows can be either regular connections or internal connections, so long as the conditions are satisfied. The length of a flow is the number of connections in the flow.

Given two ports $d_1.p$ and $d_2.q$ in G, we use the notation $\mathsf{flows}(d_1.p, d_2.q)$ for the set of flows between $d_1.p$ and $d_2.q$. If we want to restrict all the connections in the flows to be within a domain d, we use the notation $\mathsf{flows}_d(d_1.p, d_2.q)$. Note that the flows in this restricted set cannot visit any ports of d.

In Figure 7 all of the following are flows:[2]

$$\begin{array}{rcl}
\langle\cdot\rangle & \in & \mathsf{flows}(a.q, d.p) \\
\langle\cdot\rangle & \in & \mathsf{flows}(b.q, a.q) \\
\langle\cdot\rangle\ a.q\ \langle\cdot\rangle & \in & \mathsf{flows}(b.q, d.p) \\
\langle\cdot\rangle & \in & \mathsf{flows}(b.p, b.q) \\
\langle\cdot\rangle\ b.q\ \langle\cdot\rangle\ a.q\ \langle\cdot\rangle\ a.p\ \langle\cdot\rangle & \in & \mathsf{flows}(b.p, b.p) \\
\langle\cdot\rangle\ b.q\ \langle\cdot\rangle\ a.q\ \langle\cdot\rangle\ a.p\ \langle\cdot\rangle\ b.p\ \langle\cdot\rangle\ b.q\ \langle\cdot\rangle\ a.q\ \langle\cdot\rangle\ a.p\ \langle\cdot\rangle & \in & \mathsf{flows}(b.p, b.p)
\end{array}$$

The last two examples demonstrate that internal connections can make loops possible, which can lead to an infinite number of (finite) flows.

4 The Lobster Policy Language

In an information flow graph modelling a realistic SELinux system, there are likely to be common patterns of nested domains connected together, and it would be tedious to explicitly specify this repetition in the policy language. The need to capture these patterns motivates the design of Lobster, where a policy is represented as a program that, when interpreted, generates the information flow graph. As we will see in the following subsections, Lobster policies are written using an object oriented syntax: classes are instantiated to domains, and connections are like method calls.

4.1 Classes

Here is a simple Lobster class with one type and one port:

```
class C() {
  type t;
  port p : {type = t};
}
```

The `domain` statement demonstrates how to instantiate the class C into a new domain:

```
domain c = C();
```

When a class is instantiated, all the statements inside the class definition are executed, like a class constructor function in an object oriented programming language. If any of these statements creates new domains, they become nested domains.

Classes can take arguments, which are passed in when instantiating the class.

```
class D(t) { port p : {type = t}; }
domain d = D(requests);
```

[2]The notation $\langle\cdot\rangle$ is used to represent an unlabeled connection.

```
class Process() {
  port active : {position = subject};
}

class File(filenameRegex) {
  port read : {direction = output, position = object};
  port write : {direction = input, position = object};
}

class ExampleApp(dataFilenameRegexp) {
  domain app = Process();
  domain data = File(dataFilenameRegexp);
  app.active <-- data.read;
  app.active --> data.write;
}

domain example = ExampleApp("/tmp/example.*");
```

Figure 8: An example Lobster policy.

4.2 Lobster Policies to Information Flow Graphs

Lobster policies consist of a sequence of class definitions and top level `domain` and connection statements. An example Lobster policy can be seen in Figure 8.

The following algorithm is used to interpret a Lobster policy and generate an information flow graph:

1. The top level class definitions are read, and the statements within them stored.
2. The top level `domain` statements are used to instantiate some classes and create the top level domains. In this step the statements in the class definition are interpreted to build up nested structure within the domain.
3. Finally, the top level connection statements are interpreted to connect up the top level domains.

5 Compiling to SELinux Native Policies

This is how a Lobster policy is compiled into SELinux:

1. Interpret the Lobster policy to generate the information flow graph.
2. Check that the connections in the information flow graph are between compatible ports.
3. Generate the SELinux type enforcements and file contexts from the information flow graph.

5.1 Primitive SELinux Classes

For every SELinux class declared in the reference policy, there must be a Lobster class of the same name. The SELinux permissions are its ports. For example, here is an excerpt of the Lobster class corresponding to the SELinux `file` class:

```
class File(regexp) {
  port read : {direction = output};
  port write : {direction = input};
  ...
}
```

In addition to the ports resulting from their permissions, *active* classes must have a special `active` port with a `position` information flow property indicating its subject status:

```
class Process {
  port active : {position = subject}
  ...
}
```

These classes would be part of a Lobster version of the SELinux policy in force, allowing Lobster application policies to be checked in the right context.

5.2 Generating SELinux Native Policy

The connections between primitive domains define the SELinux type enforcement rules. For example, consider the following Lobster policy:

```
domain p = Process();
domain f = File("/tmp/file");
p.active -- f.read;
```

The connection would generate the SELinux type enforcement:

```
allow p_t f_t : file read;
```

It is the special `subject` information flow property of the `active` port of the `Process` class that tells the policy generator that the `p_t` type should be in the subject position and the `f_t` type should be in the object position of the `allow` rule. Exactly one of the connected ports should have the `subject` information flow property.

In addition, the special regexp argument in the instantiation of the File class would generate an SELinux file context:

```
/tmp/file  --  gen_context(system_u:domain_r:f_t,s0)
```

5.3 The Lobster Compiler

The compilation of Lobster policies into SELinux native policies has been implemented as a prototype compiler. For example, suppose the file `example.lsr` contains the example Lobster policy in Figure 8. Running the command

```
lobster example.lsr
```

results in the creation of the `example.te` output file containing the type enforcement statements

```
policy_module(example1,1.0)
type example_app_t;
type example_data_t;
allow example_app_t example_data_t:file read;
allow example_app_t example_data_t:file write;
```

and the `example.fc` output file containing the file context

```
/tmp/example.*  --  gen_context(system_u:domain_r:example_data_t,s0)
```

6 The Symbion Assertion Language

Symbion is the name of a genus of aquatic animals, less than $\frac{1}{2}$mm wide, found living attached to the bodies of cold-water lobsters.

Symbion is the name of the assertion language for information flows in Lobster policies. The idea is that flow assertions can be attached to Lobster policies; an information flow graph is then generated from the Lobster policy, and the possible flows are checked to make sure they conform with the assertions.

6.1 Expressivity

A Symbion assertion defines a set of acceptable flows between each pair of ports, and for an assertion ϕ to hold of an information flow graph G, every possible flow must be acceptable. Choosing the expressivity of the Symbion language is a trade-off between making it flexible enough to succinctly write typical information flow requirements, and making it simple enough to design efficient algorithms to check the assertions.

As a first cut, Symbion flow predicates are extended regular expressions on flows, where the atomic steps (a.k.a. letters of the alphabet) are predicates on the port or connection properties. Propositional connectives (i.e., $\wedge$, $\vee$, $\neg$ and $\Rightarrow$) are permitted to connect flow predicates. Explicit start and end anchors (i.e., `^` and `$`) are not used—the flow predicate must match the whole flow.

A Symbion assertion is a triple

$$P \to Q : \phi$$

where P and Q are predicates on the start and end ports, and ϕ is a Symbion flow predicate that defines the set of acceptable flows between them. This is what it means for this assertion to hold of an information flow graph: given any port $d.p$ that satisfies P and any port $e.q$ that satisfies Q, every flow between $d.p$ and $e.q$ must satisfy the flow predicate ϕ.

6.2 Syntax

At present there is no concrete syntax for Symbion assertions. For the purpose of showing some examples, we will use POSIX extended regular expression syntax [2], the standard symbols and precedences for propositional connectives, and informal descriptions of the atomic port and connection propositions. Port propositions are enclosed in square brackets, and connection propositions are in angle brackets.

Here are some example assertions on the information flow graph in Figure 7.

- $[a.*] \rightarrow [d.*]$: false – *"there is no flow from domain a to domain d"* – violated by the flow $a.q \; \langle\cdot\rangle \; d.p$.
- $[d.*] \rightarrow [a.*]$: false – *"there is no flow from domain d to domain a"* – this assertion succeeds.
- $[b.p] \rightarrow [b.p]$: false – *"there is no flow from port b.p to itself* – violated by the flow $b.p \; \langle\cdot\rangle \; b.q \; \langle\cdot\rangle \; a.q \; \langle\cdot\rangle \; a.p \; \langle\cdot\rangle \; b.p$.
- $[b.p] \rightarrow [b.p]$: $.* \; \langle \mathsf{isInternal} \rangle \; .*$ – *"every flow from b.p to b.p uses an internal connection* – this assertion succeeds.

Here are some examples focused on security, for hypothetical systems:

- $[secret.*] \rightarrow [internet.*]$: false – *"there is no flow from the* secret *domain to the* internet *domain"*.
- $[secret.*] \rightarrow [internet.*]$: $.* \; [\mathsf{encrypt}.*] \; .*$ – *"every flow from the* secret *domain to the* internet *domain goes through the* encrypt *domain"*.

7 Refinement

Section 6 defined the Symbion assertion language, and used it to check global properties of information flow graphs. In this section we show how it can be used to define and check refinement of information flow graphs.

7.1 Domain Specifications

A *domain specification* consists of

1. A finite set of domain ports (e.g., $\{p, q, \ldots\}$).
2. Information flow properties for each domain port.
3. A Symbion flow predicate $L_{p \rightarrow q}$ for every pair of domain ports p and q.

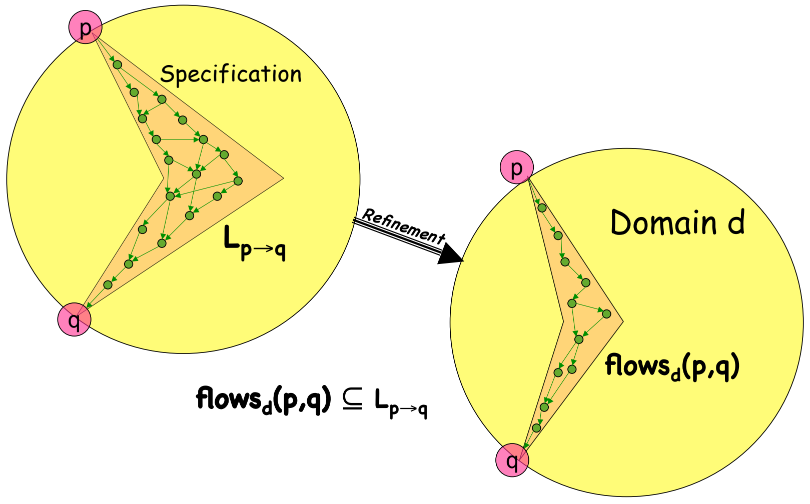


Figure 9: Verifying that the flow through the internal structure of a domain satisfies its domain specification.

7.2 Verifying Domain Specifications

Suppose we have a domain d where the internal structure is known, and a domain specification for d. It is possible to verify that the internal structure of the domain d satisfies the domain specification by using the following algorithm:

1. Check the set of domain ports exactly matches the specification.

2. Use the internal structure of d to infer a set of information flow properties for each port of d. Check that, for each port, the specified information flow properties are more specific than the inferred properties.

3. For each pair of ports p and q, compute the set of flows $\mathsf{flows}_d(p, q)$ from p to q through the internal structure of d. Check that the computed set of flows satisfies the Symbion flow predicate $L_{p \to q}$, as illustrated in Figure 9.

What does it mean for a domain to satisfy a domain specification? Step 1 ensures that the externally visible structure of the domain exactly matches the specification. Step 2 ensures that port connections that are compatible according to the domain specification are also compatible with the domain. Finally, Step 3 ensures that there are no unspecified internal flows through the domain between its ports.

7.3 Refinement

The previous section suggests a methodology for modeling information flow in complex systems. The information flow can be represented as an information flow graph, with the high-level domains having internal structure and the low-level domains as mere domain specifications.

One by one, the domain specifications can be elaborated as domains with internal structure where the lowest-level domains can be domain specifications, and the flows checked against the domain specification.

In this way, a complex system such as a computer network can be refined to an implementation in a uniform way, with some confidence that the overall security policy holds.

7.4 Primitive Domains

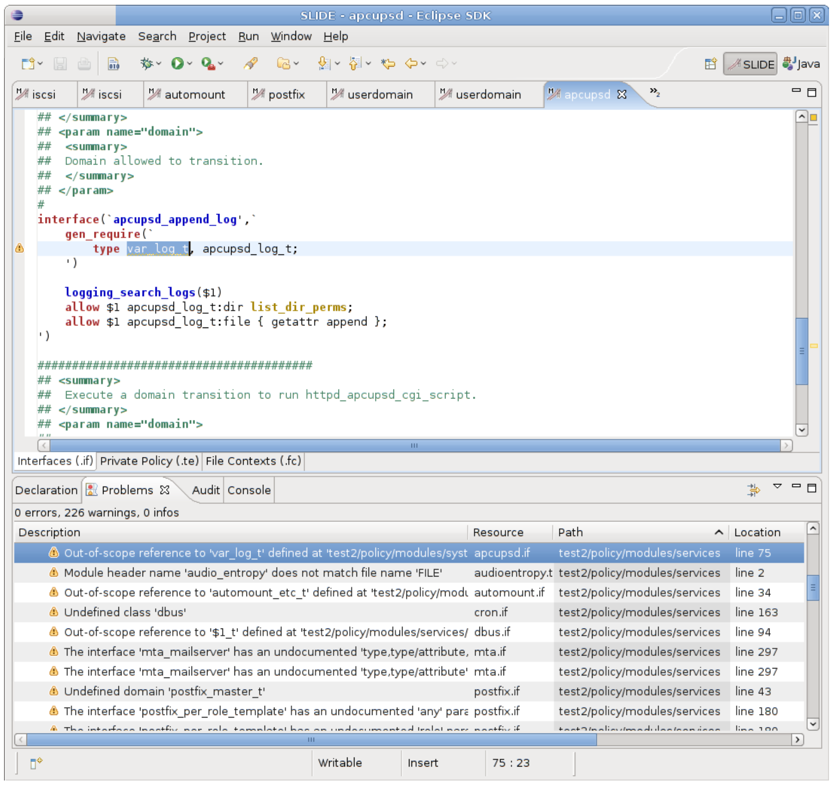
In general, it is up to the designer of the system to decide if a low-level domain has sufficient similarity to an entity on the system to regard it as primitive, and not to refine it any more.

In the case of an SELinux system, the domains can be refined down to SELinux *classes*, such as processes, files and directories. The domain specification of a file class is easy to define, by internally connecting the write port to the read port. This captures the fact that a file is a passive storage mechanism, and any data that is written to the file can later be read.

The case of a process class is more complicated. The process is an active subject, and may read data from multiple inputs and write to multiple outputs, without it necessarily being the case that there is an information flow from every input to every output. For most processes, it will not matter if this worst-case scenario is assumed; for certain security-critical processes, it may require a formal proof that there is no information flow from one input to one output.

8 Analyzing SELinux Reference Policies using Shrimp

Through the introduction of modular policies with separation of interface and implementation, the Reference Policy [9] has enabled SELinux policies to be developed and maintained in a way that scales to large Linux distributions with hundreds of packages. The Reference Policy has had a clear impact in that it is included in several large distributions, including Red Hat Enterprise Linux, Fedora, Gentoo and Debian. As such, the Reference Policy is a success in applying software-engineering principles to solve the problem of writing and maintaining large security policies. In its current tool incarnation, many of the principles behind the Reference Policy are supported only by convention and not actually machine checked. One reason for the limited automatic checks is the fact that there is almost no analysis taking place when translating Reference Policy modules to native SELinux modules. This is due to the use of standard text-processing tools such as m4, which are convenient for developing straightforward macro-expansion transformations, but fall short in providing support for non-local analysis.



```
## </summary>
## <param name="domain">
##  <summary>
##  Domain allowed to transition.
##  </summary>
## </param>
#
interface(`apcupsd_append_log',`
    gen_require(`
        type var_log_t, apcupsd_log_t;
    ')

    logging_search_logs($1)
    allow $1 apcupsd_log_t:dir list_dir_perms;
    allow $1 apcupsd_log_t:file { getattr append };
')

########################################
## <summary>
##  Execute a domain transition to run httpd_apcupsd_cgi_script.
## </summary>
## <param name="domain">
```

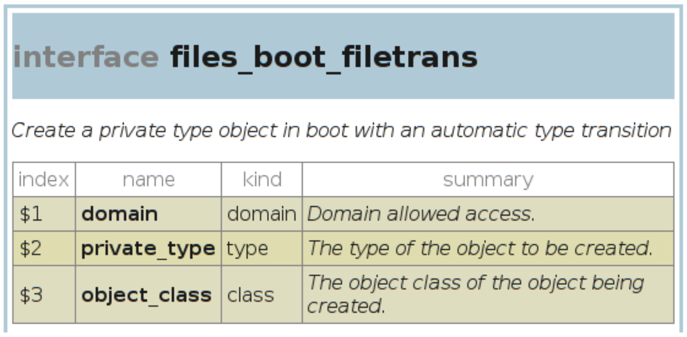
0 errors, 226 warnings, 0 infos

Description	Resource	Path	Location
Out-of-scope reference to 'var_log_t' defined at 'test2/policy/modules/syst	apcupsd.if	test2/policy/modules/services	line 75
Module header name 'audio_entropy' does not match file name 'FILE'	audioentropy.t	test2/policy/modules/services	line 2
Out-of-scope reference to 'automount_etc_t' defined at 'test2/policy/modu	automount.if	test2/policy/modules/services	line 34
Undefined class 'dbus'	cron.if	test2/policy/modules/services	line 163
Out-of-scope reference to '$1_t' defined at 'test2/policy/modules/services/	dbus.if	test2/policy/modules/services	line 94
The interface 'mta_mailserver' has an undocumented 'type,type/attribute,	mta.if	test2/policy/modules/services	line 297
The interface 'mta_mailserver' has an undocumented 'type,type/attribute'	mta.if	test2/policy/modules/services	line 297
Undefined domain 'postfix_master_t'	postfix.if	test2/policy/modules/services	line 43
The interface 'postfix_per_role_template' has an undocumented 'any' parа	postfix.if	test2/policy/modules/services	line 180

Figure 10: Shrimp integrated into SLIDE from Tresys. The top window is the standard SLIDE view of a policy module. The bottom part shows a Problems tab with errors detected by Shrimp.

Particularly given the impact that the Reference Policy has, it is valuable to have tools that could could analyze policy modules before macro expansion, by treating the Reference Policy as a domain-specific policy language, as it was intended. We developed the analysis tool *Shrimp* for this purpose. Shrimp treats many of the low-level macros in the Reference Policy as language-level keywords, and this allows us to mechanically check many of the rules that used to be "by convention" and never really enforced. For example, Shrimp detects when one module breaks the abstraction barrier and directly refers to a type in another module. Roughly, Shrimp is to the Reference Policy what Lint [4] is to C.

A team at Tresys lead by David Sugar has developed a plugin that integrates Shrimp into SLIDE [13], which is an integrated development environment for SELinux policy modules, see Figure 10. This has increased the usefulness of Shrimp—together, Shrimp and SLIDE can not only detect policy errors but also suggest corrections.



interface files_boot_filetrans

Create a private type object in boot with an automatic type transition

index	name	kind	summary
$1	**domain**	domain	*Domain allowed access.*
$2	**private_type**	type	*The type of the object to be created.*
$3	**object_class**	class	*The object class of the object being created.*

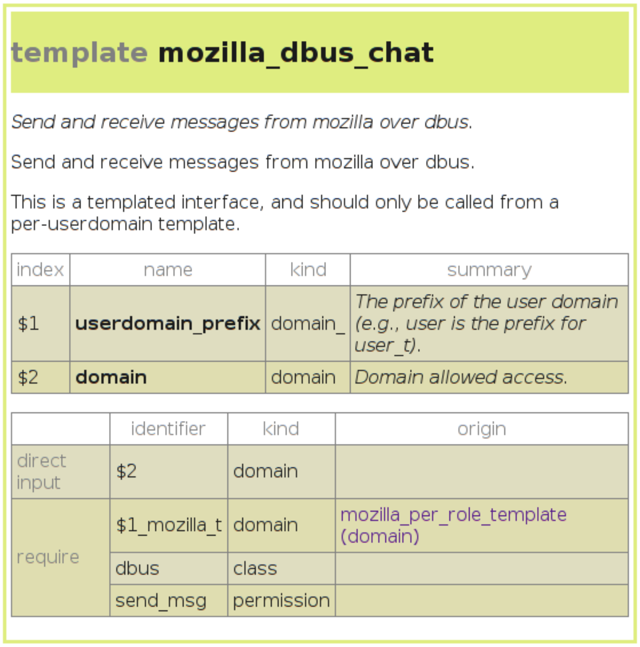
Figure 11: Excerpt from documentation generated for an interface in the Reference Policy. The third column contains type information (called kind information since "type" has a special meaning in SELinux) that has been inferred by Shrimp. The other columns are taken from documentation provided by the policy writer.

Based on these basic capabilities, we extended Shrimp to perform type inference on Reference Policy macros. We use the result of this analysis to produce HTML documentation of the Reference Policy that includes inferred types of parameters, see Figure 11.

Shrimp also infers where symbols are declared (exported) and required (imported). An example is illustrated in Figure 12, where it has been inferred that the template (one type of macro in the Reference Policy) `mozilla_dbus_chat` requires a domain symbol that is formed by appending `_mozilla_t` to its first parameter. This symbol can be declared by calling the macro `mozilla_per_role_template`, which Shrimp reflects in the origin column. This reveals a dependency between macros, in that the dbus-chat template can only be called after the per-role template has been called. In fact, this dependency suggests that the per-role template could be seen as a method on an object created by the dbus-chat template. Based on Shrimp, we are currently prototyping a translation tool from Reference Policy modules into Lobster policies, and are investigating to what extent the template dependency analysis can be used to give the generated policies more structure.

9 Use Case: Policies for Guards

Although the Lobster compiler is still under development, we have been able to apply it in the Guardol program by Rockwell Collins. In this work, we generate information flow policies in Lobster from guard specifications, and then translate them into SELinux policies that confine the components of the guard. We also see an opportunity to use Symbion assertions to state explicit restrictions in how guard components may interact. By introducing high-level


template mozilla_dbus_chat

Send and receive messages from mozilla over dbus.

Send and receive messages from mozilla over dbus.

This is a templated interface, and should only be called from a per-userdomain template.

index	name	kind	summary
$1	**userdomain_prefix**	domain_	*The prefix of the user domain (e.g., user is the prefix for user_t).*
$2	**domain**	domain	*Domain allowed access.*

	identifier	kind	origin
direct input	$2	domain	
require	$1_mozilla_t	domain	mozilla_per_role_template (domain)
	dbus	class	
	send_msg	permission	

Figure 12: Shrimp analysis results for the template `mozilla_dbus_chat`.

information flow specifications with assertions we get additional opportunities for both human and automatic correctness checks. We believe these checks together help us gain assurance that guards behave according to specification.

10 Related Work

The Reference Policy [9] from Tresys raises the level above the SELinux native policy by introducing the concept of modules that separate interface from implementation of a security policy for a software package. This makes it possible for programmers to define at an abstract level how a piece of software interacts with the rest of the system by putting together a policy module that calls interfaces in existing modules. However, the Reference Policy doesn't directly support the need for understanding what the information flow is among a set of modules. Part of the problem is that the Reference Policy is not precisely defined since it is implemented using the general-purpose macro processor m4, which does not enforce any of the intended abstraction barriers. Another reason is more fundamental, in that one has to look inside the implementation of every module to figure out what information flows it entails. We describe our effort in analyzing the Reference Policy language in Section 8.

In developing an alternative to the Reference Policy called SEng [5], Kuliniewicz recognized the problem of analyzing policies that are defined in terms of low-level macro processors. SEng is modeled closely after the Reference Policy, but eliminates the use of m4 macro definitions in favor of several language primitives for different kinds of abstractions.

PLEASE is another language presented by Quigley as an alternative to the Reference Policy [10], based on an object-oriented design. Security domains are called resources, and such resources may export permissions (which resemble methods) and extend other resources (resembling inheritance). Quigley's work is an indicator that an object-oriented point of view is helpful in designing security-policy languages, and provides us with additional motivation to look for an implicit object-oriented structure in the design of the Reference Policy.

The Polgen Specification Language (PSL) [11] is based on declarations of components corresponding to security domains. Each component declaration contains a list of connectors to other components. There is no way to define what kind of access is granted to a component in its declaration—the only way to find out is to examine all other components in the system for connectors that refer to it.

There are a number of useful tools that can analyze SELinux policies on the native or binary levels. These include setools, apol, Gokyo, and SLAT [14, 12, 3, 1]. It would be useful to re-render some of these analyses at the Reference Policy level based on Shrimp.

Similar to Lobster, CDSFramework [6] uses hierarchical domains to design SELinux policies. Differences include the distinction between resources (passive) and domains (active), and unlimited access between all entities within a domain. Whereas Lobster is based on a general notion of security domains and information flows, CDSFramework's focus is on providing a higher level of abstraction on top of the Reference Policy, and a dictionary-based translation mechanism into Reference Policy modules. This provides an easier path to compiling down to working policies, but complicates the analysis of the higher level policy language.

11 Summary

By using the domain-specific languages Lobster and Symbion, we can express security policies based on information flows between nested security domains, and we have shown how these can be used to specify high-level policies that can be compiled into SELinux policies. We also introduced the analysis tool Shrimp, which appears to be fairly unique in that it allows us to analyze the SELinux Reference Policy language without first expanding away its macro definitions. We hope Shrimp may become a useful tool for increasing the quality of the Reference Policy.

We see a number of ways to bring this work forward, and we will outline them in the next section.

12 Future Work

Going forward, we see opportunities to further strengthen SELinux target support for Lobster and Symbion. This support is bidirectional in that we are interested both in the generation and analysis of SELinux policies.

12.1 Targeting the SELinux Reference Policy

Currently, the Lobster compiler produces native SELinux policy modules that cannot refer to other modules in the Reference Policy. We are investigating compilation schemes that would allow us to better integrate with the Reference Policy. As a first step, we would allow Lobster policies to refer to existing Reference Policy modules. Although this may simplify practical policy development, it may not be a straightforward task since we want to avoid introducing hidden information flows in the process. Analysing the Reference Policy with Shrimp and reverse compiling it into Lobster may help here.

12.2 Reverse Compiling the Refererence Policy into Lobster

We have initial prototypes that can reverse compile the SELinux Reference Policy into Lobster policies. By increasing the fidelity of our reverse-compilation method, we see opportunities not only to understand existing SELinux policies in terms of Lobster, but also a way to better interface between new Lobster policies and an existing SELinux platform.

12.3 Visualization and Use Cases

Another major effort is to provide visualizations of security policies from Lobster. To understand large and complicated policies, we believe that good visualization tools will be essential. Together with reverse-compilation tools from SELinux and other domains, Lobster visualization tools will help users understand existing, large security policies.

We plan to find more SELinux-based use cases for Lobster. Candidates can be found in other Galois technologies like the Trusted Services Engine (TSE), the ML-Wiki, and cross-domain RSS solutions. All of these technologies may benefit from having high-level descriptions of their information flow properties. Moreover, these technologies can potentially be deployed on SELinux platforms where assurance can be gained by use of enforcing SELinux policies that we can generate automatically.

We also plan to continue to develop the Guardol use case in Section 9 together with Rockwell Collins.

12.4 Future work beyond the SELinux Domain

We believe the concepts of nested security domains, information flows and assertions that Lobster and Symbion provide are general enough that these languages can be used to describe and analyze security policies for systems beyond SELinux.

A particularly good fit may be Secure Virtual Platforms where some security policies describe the interior of virtual machines, using SELinux for example, and another policy describes the interaction between the virtual machines through the hypervisor, using Xen Security Modules. With Lobster, we can have one coherent description of the information flow policy for the whole platform. We can describe each virtual machine as its own nested security domain which may be further refined. By using separate back ends, we can compile the overall policy down to separate SELinux policies and a hypervisor policy.

A more long-term vision is to use Lobster to explicitly capture information flow properties in larger systems comprising of networks of computers, and perhaps complete organizations. In this vision (called *Nova Scotia*, where the world's largest lobster was caught), we would have an open standard against which we and others can develop backends for compiling back and forth between Lobster and various policy and configuration languages.

Having one comprehensive language for expressing security policies for large systems would allows us to capture and check more security properties. For example, we would be able to express that an organization should have firewalls on every connection between its internal network and the Internet.

References

[1] Amy L. Herzog and Joshua D. Guttman. Achieving security goals with security-enhanced linux. Technical report, The MITRE Corporation, 2002.

[2] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, 2001.

[3] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[4] S. C. Johnson and S. C. Johnson. Lint, a c program checker. In *Comp. Sci. Tech. Rep*, pages 78–1273, 1978.

[5] Paul Kuliniewicz. SENG: An enhanced policy language for SELinux. Security Enhanced Linux Symposium, 2006.

[6] Karl MacMillan, Spencer Shimko, Chad Sellers, Frank Mayer, and Art Wilson. Lessons learned developing cross-domain solutions on SELinux. SELinux Symposium, 2006.

[7] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example*. Open Source Software Development Series. Prentice Hall, 2007.

[8] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly, 2005.

[9] Christopher J. PeBenito, Frank Mayer, and Karl MacMillan. Reference policy for security enhanced linux. Security Enhanced Linux Symposium, 2006.

[10] D. P. Quigley. PLEASE: Policy Language for Easy Administration of SELinux. Master's thesis, Stony Brook University, May 2007. Technical Report FSL-07-02, `www.fsl.cs.sunysb.edu/docs/dquigley-msthesis/dquigley-msthesis.pdf`.

[11] Brian T. Sniffen, David R. Harris, and John D. Ramsdell. Guided policy generation for application authors. SELinux Symposium, 2006.

[12] Tresys Technology. *Apol Help Files*, 2008. Available from `http://oss.tresys.com/projects/setools/wiki/helpFiles/iflow_help`.

[13] Tresys Technology. *SELinux Policy IDE – SLIDE*, 2009. Available from `http://oss.tresys.com/projects/slide`.

[14] Tresys Technology. *SETools - Policy Analysis Tools for SELinux*, 2009. Available from `http://oss.tresys.com/projects/setools`.