Security Configuration Domain Specific Language (DSL)
SELinux Developers Summit
Ottawa 2008

Peter White

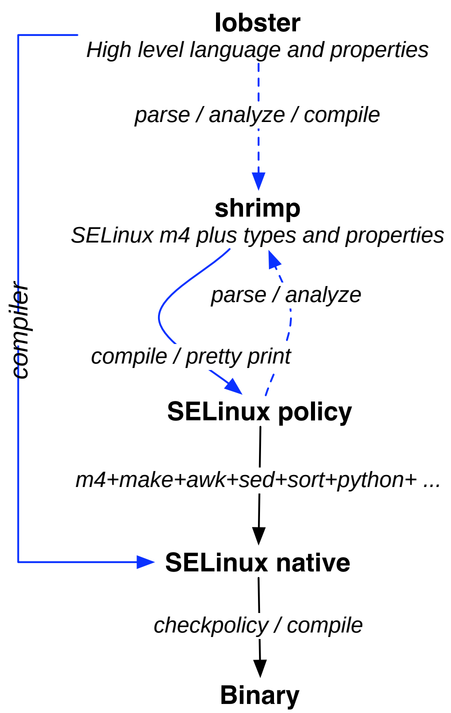
galois

Outline

- Policy DSL objectives
- Project architecture
- Shrimp: Reference policy with ~~type~~ kind checking
- Lobster: Higher order policy language

Project objectives

- Shrimp: Improve the reference policy language by moving into a formal setting
  - Address issues such as
    - Wrong number of parameters
    - Duplicate macros
    - Call to undefined macro
- Lobster: Provide abstractions that alleviate the tedium and detail of specifying a SELinux security policy



Language trajectory

Language	Manipulates	Fancier stuff	Semantics	Value added
Native	Permissions Files	Sets of permissions, roles, constraints	Relational	Much better than writing binary
Reference	Modules interfaces files	Macros	Native Modules as global vars	Modularity
Shrimp	Modules interfaces files	Macros and kinds	Declarative modules	Enforcement of modularity
Lobster	Objects classes methods	TBD	Flow graph Native Shrimp	Abstraction

Shrimp

Purposes and possibilities of Shrimp

- Support for analysis of Reference Policy on its own level - not in terms of Native Policy.
- ``lint'' tool for Reference Policy.
- HTML generation of documentation + analysis results for Reference Policy.
- Prototyping workbench for a new Reference Policy language ``Shrimp''.
- Target for Lobster compilation.
- Conversion tool from Reference Policy to Shrimp (future)

Shrimp anatomy

- *kind information* for interface parameters
  - The *kind* system is actually a *type* system in programming language parlance - we attempt to avoid overloading the word *type*
- Local and global *information-flow properties* (future)

A kind system for Shrimp

- Statement judgments for Reference Policy statements are of the form: $\Gamma \mid s :: R;O$, which reads
  - "Given a symbol environment , statement s demands the the symbols *R* are provided by the policy, and puts the symbols in *O* into the policy"
- Example: $\Gamma \mid$ `type t` $:: \varnothing; t : type$
  - "the statement `type t' puts the type t into the policy"
- Composition of statements: The *R* and *O* demands enrich the symbol environment for later statements:

$$\frac{\Gamma \mid\!- s_1 :: R_1;O_1 \qquad \Gamma \mid\!- s_1 :: R_1;O_1 \qquad O_1 \text{ and } O_2 \text{ disjoint}}{\Gamma \mid\!- s_1;s_2 :: R_1 \cup R_2;\ O_1 \cup O_2}$$

“Lint” results from kind analysis

```
Undefined identifiers: [{
../Reference-Policy/refpolicy/policy/modules/kernel/kernel.if:1014:32:proc_t,[type/attribute]}]
(100 errors like this.)
Mismatch between number of documented vs. referenced parameters:
## <param name="domain" />
## <param name="userdomain_prefix" />
## <param name="domain" />
[{\$1,[attribute_]}, {\$2,[type]}]
(29 errors like this.)
Wrong number of arguments: {
../Reference-Policy/refpolicy/policy/modules/apps/java.if:210:9:userdom_unpriv_usertype,
  [[attribute_], [type], [any]]}
(19 errors like this.)
Call to undefined macro:
../Reference-Policy/refpolicy/policy/modules/system/userdomain.if:202:17:fs_read_nfs_named_sockets
(10 errors like this.)
Duplicate definition of macro:
../Reference-Policy/refpolicy/policy/support/obj_perm_sets.spt:334:9:all_nscd_perms
(5 errors like this.)
Illegal symbol declarations in interface: [
../Reference-Policy/refpolicy/policy/modules/kernel/selinux.if:514:14:\$1]
Duplicate definition of (
../Reference-Policy/refpolicy/policy/modules/kernel/corenetwork.te:1533:25:netif_lo_t,type)
```

template dbus_user_bus_client_template

Template for creating connections to a user DBUS.

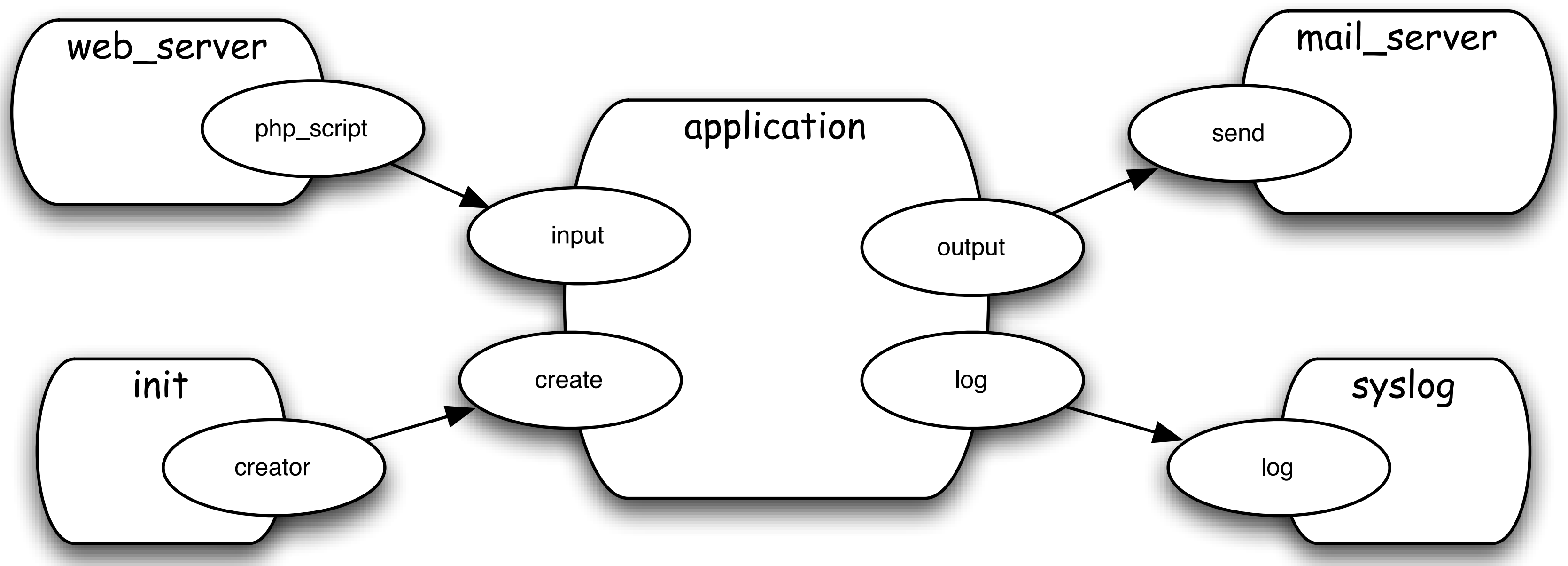
index	name	kind	summary
$1	**user_prefix**	domain_	*The prefix of the domain (e.g., user is the prefix for user_t).*
$2	**domain_prefix**	domain_	*The prefix of the domain (e.g., user is the prefix for user_t).*
$3	**domain**	domain	*The type of the domain.*

	identifier	kind	origin
input	$3	domain	
origin	$2_dbusd_$1_t	type	
require	$1_dbusd_t	type	dbus_per_role_template (type) *userdom_restricted_xwindows_user_template (type)*
	dbus	class	
	send_msg	permission	

Kind inference results as HTML

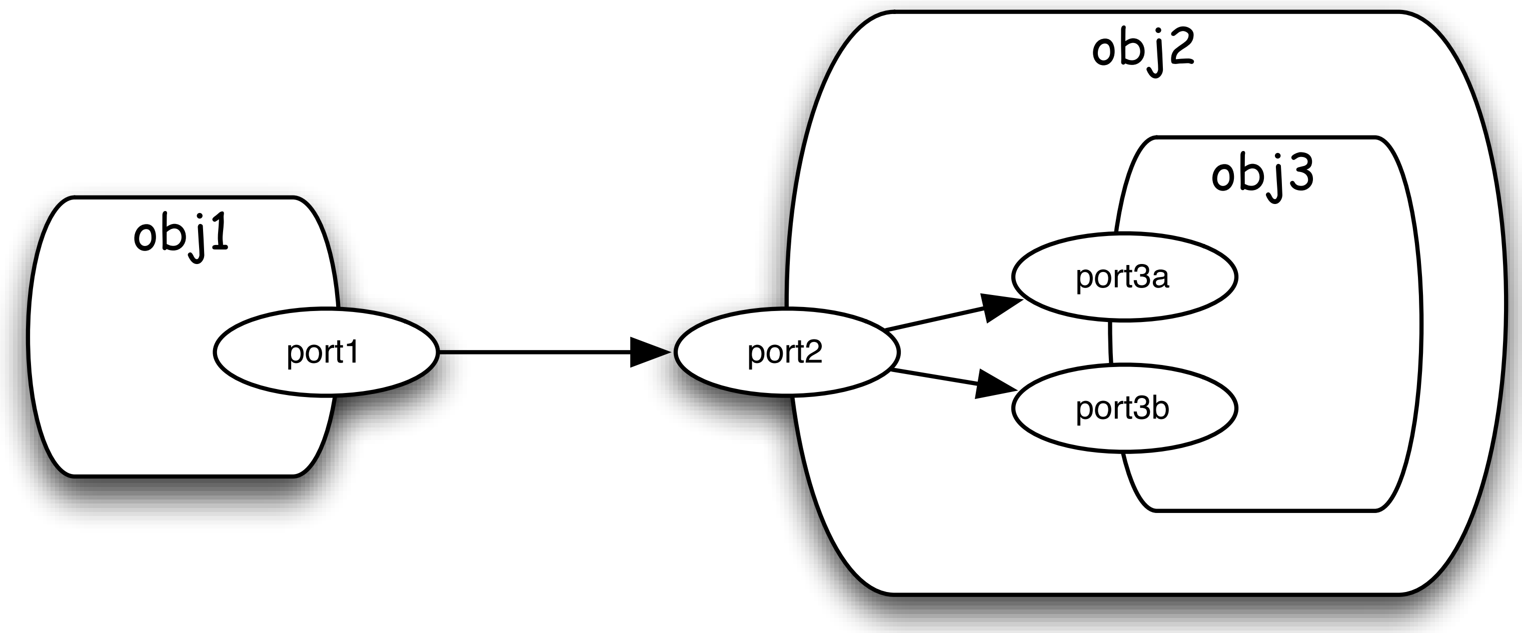
Lobster

direction of arrows shows information flow



Security policy designer's view

provides one basis for abstraction in Lobster



Objects can be nested

Lobster use case

- This is the intended use of the Lobster DSL
  - A security policy designer writes a Lobster information flow diagram for the application
  - A developer write a Lobster policy for the application
  - An automatic tool verifies that the Lobster policy is a refinement of the Lobster information flow diagram, in that no extra information flows have been introduced
  - A compiler takes the Lobster policy and generates SELinux policy statement
    - In Shrimp
    - In SELinux Native policy

Lobster snippet

```
class F (path, level) {
  process = new "F" Process;
  port write : { type = X };
  port read  : { type = X };
  port executable  -- process.active;
  port create      -- process.transition;

  f = new "f" SimpleFile ( X, path );
  write  --> f.write;
  read   <-- f.read;
}
```

Goal: Lobster bisque and shrimp cocktail

Next steps

- Incorporation of Shrimp into SLIDE tools
- Elevate Shrimp from lint to language
- Provide graphical front end for Lobster
- Add abstraction capabilities to Lobster informed by trials on real systems
  - Work to reduce tedium and repetition
- Add high level policy constraints to Lobster
  - e.g. "Process A can communicate to process C only via the intermediary B"
- Add trust annotations to objects, in support of overall system certification

End

|galois|

Backup slides

The symbol environment

- The symbol environment Γ is local to macro definitions and implementation modules, and it is consulted in e.g. access-rule statements:
  - If the symbol environment determines that s is a domain, *t* is a type or an attribute, c is a class and p is a permission, then we can say that the statement `allow s t: c p` is permissible, without any interaction with the policy

$$\frac{\texttt{Γ,s:domain,t:type,c:class,p:permission|-}}{\texttt{allow s t:c p :: ∅;∅}}$$

Reference policy source

```
##############################################
## <summary>
##      Template for creating connections to
##      a user DBUS.
## </summary>
## <param name="user_prefix">
##      <summary>
##      The prefix of the domain (e.g., user
##      is the prefix for user_t).
##      </summary>
## </param>
## <param name="domain_prefix">
##      <summary>
##      The prefix of the domain (e.g., user
##      is the prefix for user_t).
##      </summary>
## </param>
## <param name="domain">
##      <summary>
##      The type of the domain.
##      </summary>
## </param>
#
template(`dbus_user_bus_client_template',`
        gen_require(`
                type $1_dbusd_t;
                class dbus send_msg;
        ')

#       type $2_dbusd_$1_t;
#       type_change $3 $1_dbusd_t:dbus $2_dbusd_$1_t;

        # SE-DBus specific permissions
#       allow $2_dbusd_$1_t { $1_dbusd_t self }:dbus send_msg;
        allow $3 { $1_dbusd_t self }:dbus send_msg;

        # For connecting to the bus
        allow $3 $1_dbusd_t:unix_stream_socket connectto;
')
```

Primitive classes

- For every SELinux class, there must be a Lobster class of the same name. The SELinux permissions are its ports

```
class File( regexp ) {
  port getattr : { type = x };
  port read    : { type = x };
}
```

- 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.

Language hierarchy