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

Peter White
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

<table>
<thead>
<tr>
<th>Language</th>
<th>Manipulates</th>
<th>Fancier stuff</th>
<th>Semantics</th>
<th>Value added</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>Permissions Files</td>
<td>Sets of permissions, roles, constraints</td>
<td>Relational</td>
<td>Much better than writing binary</td>
</tr>
<tr>
<td>Reference</td>
<td>Modules interfaces files</td>
<td>Macros</td>
<td>Native Modules as global vars</td>
<td>Modularity</td>
</tr>
<tr>
<td>Shrimp</td>
<td>Modules interfaces files</td>
<td>Macros and kinds</td>
<td>Declarative modules</td>
<td>Enforcement of modularity</td>
</tr>
<tr>
<td>Lobster</td>
<td>Objects classes methods</td>
<td>TBD</td>
<td>Flow graph Native Shrimp</td>
<td>Abstraction</td>
</tr>
</tbody>
</table>
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: $\Gamma \vdash s :: R;O$, which reads
  • “Given a symbol environment $\Gamma$, statement $s$ demands that the symbols $R$ are provided by the policy, and puts the symbols in $O$ into the policy”

• Example: $\Gamma \vdash \text{type } t :: \emptyset; t : \text{type}$
  • “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:

\[
\begin{align*}
\Gamma \vdash s_1 :: R_1; O_1 & \quad \Gamma \vdash s_1 :: R_1; O_1 \quad \text{and } O_1 \text{ and } O_2 \text{ disjoint} \\
\Gamma \vdash s_1; s_2 :: R_1 \cup R_2; O_1 \cup O_2
\end{align*}
\]
“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.

<table>
<thead>
<tr>
<th>index</th>
<th>name</th>
<th>kind</th>
<th>summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td><code>user_prefix</code></td>
<td>domain_</td>
<td>The prefix of the domain (e.g., user is the prefix for user_t).</td>
</tr>
<tr>
<td>$2</td>
<td><code>domain_prefix</code></td>
<td>domain_</td>
<td>The prefix of the domain (e.g., user is the prefix for user_t).</td>
</tr>
<tr>
<td>$3</td>
<td><code>domain</code></td>
<td>domain_</td>
<td>The type of the domain.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>identifier</th>
<th>kind</th>
<th>origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>$3</td>
<td>domain</td>
</tr>
<tr>
<td>origin</td>
<td>$2_dbusd_$1_t</td>
<td>type</td>
</tr>
<tr>
<td>require</td>
<td>$1_dbusd_t</td>
<td>type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dbus</code></td>
<td>class</td>
</tr>
<tr>
<td></td>
<td><code>send_msg</code></td>
<td>permission</td>
</tr>
</tbody>
</table>
direction of arrows shows information flow

web_server
php_script
application
input
output
init
creator
mail_server
send
syslog
log
create
log

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 writes 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 statements
    - 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;
}

© 2008 Galois, Inc.
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
Backup slides
The symbol environment

- The symbol environment $\Gamma$ 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 $\text{allow } s \ s t: c \ p$ is permissible, without any interaction with the policy

\[
\Gamma, s:\text{domain}, t:\text{type}, c:\text{class}, p:\text{permission} |- \\
\text{allow } s \ t: c \ p :: \emptyset; \emptyset
\]
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 name="domain_prefix">
<summary>
The prefix of the domain (e.g., user is the prefix for user_t).
### </summary>
### <param name="domain">
<summary>
The type of the domain.
### </summary>
#
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

```plaintext
class File( regexp ) {
    portgetattr : { type = x };
    portread : { 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

- Higher level policy language
  - Permission
  - Permission-Expression
  - Lifted Correspondence
  - Action-Expression
  - Method

- Middle level policy language
  - Permission
  - Permission-Expression
  - Lifted Correspondence
  - Interface-Expression
  - Module interface

- Low level policy language
  - Permission
  - Permission-Expression
  - Lifted Correspondence
  - action-Expression
  - action

Correspondence manipulates controls

Compile derived translation derived implementation