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

Magnus Carlsson, Joe Hurd, Sigbjorn Finne, Brett Letner,
Joel Stanley, Louis Testa, Peter White

Galois, Inc.

May 18, 2009

Overall goals

- Security policies that we can understand
  - Domain-specific, high-level language: nested security domains, information flows and assertions
- Established links with existing, relevant policy languages
  - SELinux
- Long term: Descriptions of large, heterogeneous systems
  - Networks with guards, firewalls, virtual machines,...

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

Status and future

Conclusion

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

Status and future

Conclusion

Background - SELinux

- NSA: put Mandatory Access Control (using the FLASK architecture) into Linux (Loscocco, Smalley, 2001)
- Released with *Example Policy*
- Fine-grained control over allowed permissions
- The *native policy language*: limited support for abstraction and re-use

SELinux Native Policy Sample

Allow init to start httpd

```
allow httpd_t init_t:process sigchld;
allow httpd_t init_t:process signull;
allow httpd_t httpd_exec_t:file entrypoint;
allow httpd_t httpd_exec_t:file { getattr read execute };
allow httpd_t httpd_exec_t:file { ioctl lock };
typeattribute httpd_exec_t entry_type;
typeattribute httpd_exec_t exec_type;
typeattribute httpd_exec_t file_type;
role system_r types httpd_t;
allow initrc_t httpd_exec_t:file { getattr read execute };
allow initrc_t httpd_t:process transition;
dontaudit initrc_t httpd_t:process { noatsecure siginh };
dontaudit initrc_t httpd_t:process rlimitinh;
type_transition initrc_t httpd_exec_t:process httpd_t;
allow httpd_t initrc_t:fd use;
allow httpd_t initrc_t:fifo_file { getattr read write };
allow httpd_t initrc_t:fifo_file { append ioctl lock };
allow httpd_t initrc_t:process sigchld;
```

Background - SELinux

- Re-use improved by using *macro definitions*
- Policy code on previous slide captured with one macro call:

```
init_daemon_domain(httpd_t,httpd_exec_t)
```

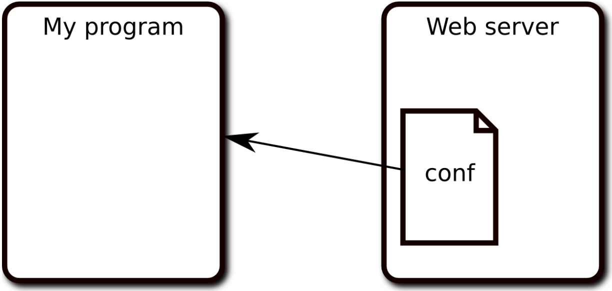
- Tresys Technology: introduced the *Reference Policy* (PeBenito, Mayer, MacMillan, 2006)
  - Modularized the Example Policy
  - Separated interface from implementation
  - Discouraged the unstructured use of “global variables”
  - Successful application of software engineering principles
  - Large impact - used in a number of wide-spread Linux distributions
  - Complex - over 150,000 lines in 250 modules

SELinux - Reference Policy example

Separation of interfaces from implementation:

```
apache_read_config(my_program_t)
```

allows my program to access the configuration files of the web server



SELinux - the Reference Policy

- The Reference Policy: enabled security policies to scale
- Issues:
  - Difficult to analyze - e.g. what are the information flows between modules?
  - Still implemented using textual processing (macro-expansion)

Problem with macro processors

- Macro processors provide simple means of defining domain-specific languages by defining language "primitives" as macro definitions
- Problem: limited means for analyzing the input
  - checking types and number of parameters
  - controlling where macros should be expanded

Macro processing in the Reference Policy

- Example: processing of *file contexts*:

```
/bin/login -- gen_context(system_u:object_r:login_exec_t,s0)
```

- Associates the login program with the security domain `login_exec_t`

Macro processing in the Reference Policy

- Example: processing of *file contexts*:

```
/bin/login -- gen_context(system_u:object_r:login_exec_t,s0)
```

- Associates the login program with the security domain `login_exec_t`
- `gen_context` is a macro, and the specification is expanded to

```
/bin/login  -- system_u:object_r:login_exec_t
```

Macro processing in the Reference Policy

- Suppose we want to specify that a new program we call `secure_mode_conf` should be in the security domain `my_domain`:

```
/bin/secure_mode_conf -- gen_context(system_u:object_r:my_domain,s0)
```

Macro processing in the Reference Policy

- Suppose we want to specify that a new program we call `secure_mode_conf` should be in the security domain `my_domain`:

```
/bin/secure_mode_conf -- gen_context(system_u:object_r:my_domain,s0)
```

- Problem: `secure_mode_conf` happens to be a macro definition, and the text expands to:

```
/bin/false -- system_u:object_r:my_domain
```

- Accidentally, we have associated the system program `false` to be in our security domain!
- The Reference Policy has over 500 macros that may clash

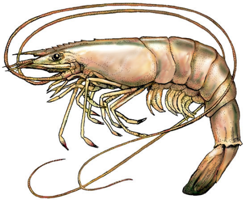
Macro processing in the Reference Policy

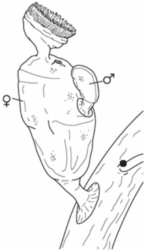
- Macro processing makes it difficult to understand the Reference Policy
- Current analysis tools work on policy after macro expansion
- Policy writers that use analysis tools must understand policy languages both before and after macro expansion

Challenges with SELinux

- How can the Reference Policy be analyzed without macro expansion?
- How can we understand information flow without looking inside policy modules?
- How can we explicitly state restrictions in information flows between modules?

Approach





Shrimp — Treating the Reference Policy as a domain-specific language in its own right

- Giving a precise specification of the Reference Policy language

Lobster — Policy language based on *information flow* and *nested security domains*

Symbion — Assertion language over information flows and domains

- example: “every flow from the Secret domain to the Internet domain goes through the Encryption domain.”

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

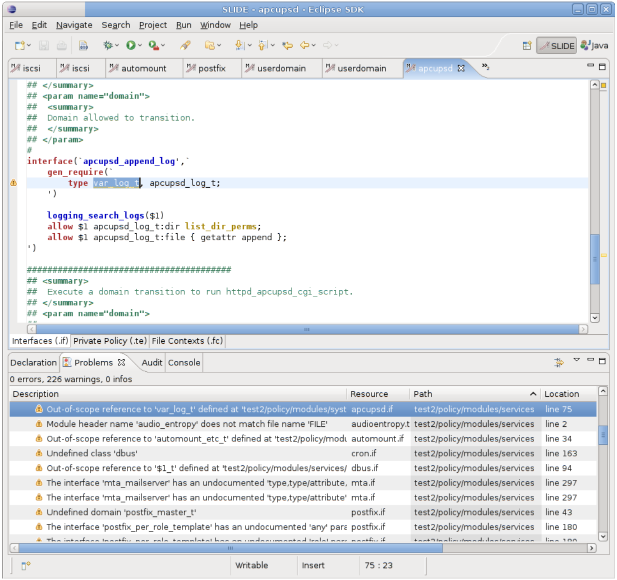
Status and future

Conclusion

Shrimp

- Treating the SELinux Reference Policy as a domain-specific language in its own right
- Gives a precise specification of the Reference Policy semantics (collaboration with Tresys)
- Allows us to analyze the complete Reference Policy and detect over 100 problems ("lint" for policies)
- Example: illegal references to private types across module boundaries (the equivalent of "global variables")
- Integrated with SLIDE (an IDE for policy writers) (David Sugar & co, Tresys Technology)
- Our hope: Shrimp will help increase our confidence in the Reference Policy

Shrimp in SLIDE



```
## </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' para	postfix.if	test2/policy/modules/services	line 180
⚠ The interface 'postfix_per_role_template' has an undocumented 'role' para	postfix.if	test2/policy/modules/services	line 180

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

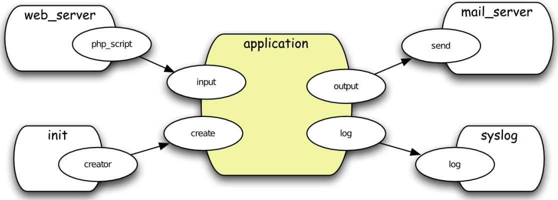
Experience

Status and future

Conclusion

A Security Policy Designer’s View

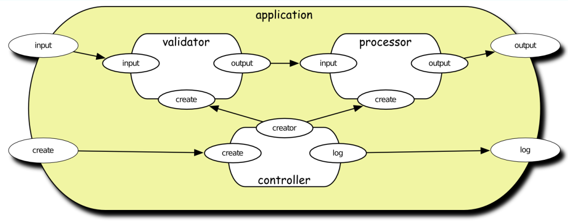
How a security policy designer might see an application:



Security domains with explicit information flow through ports

Security Policy Designer's View

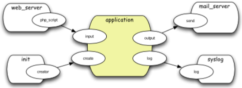
Inside the application, there might be some more information flow specified:



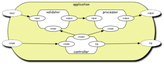
Lobster Use Case

The intended use of Lobster:

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



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

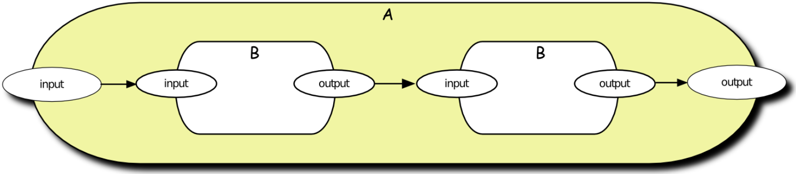


3. An automatic tool verifies that the Lobster policy (2) is a refinement of the Lobster information flow graph (1), in that no extra information flows have been introduced.
4. A compiler takes the Lobster policy (2) and generates SELinux policy statements.

Information flow graphs in Lobster

```
class A() {
  port input;
  port output;
  domain p = B();
  domain q = B();
  input --> p.input;
  p.output --> q.input;
  q.output --> output;
}
```

```
class B() {
  port input;
  port output;
}
```



Compiling Lobster to SELinux

Primitive classes have ports corresponding to SELinux permissions:

```
domain d = Process();
domain f = File("/etc/foo");
d.active <-- f.read;
d.active --> f.write;
```

gets translated to

```
allow d_t f_t:file { read write };

/etc/foo -- gen_context(system_u:object_r:f_t,s0)
```

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

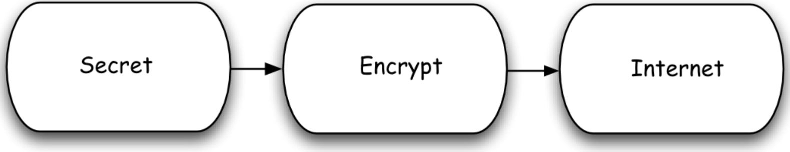
Status and future

Conclusion

Background: Assertions on flows

Simple on small policies

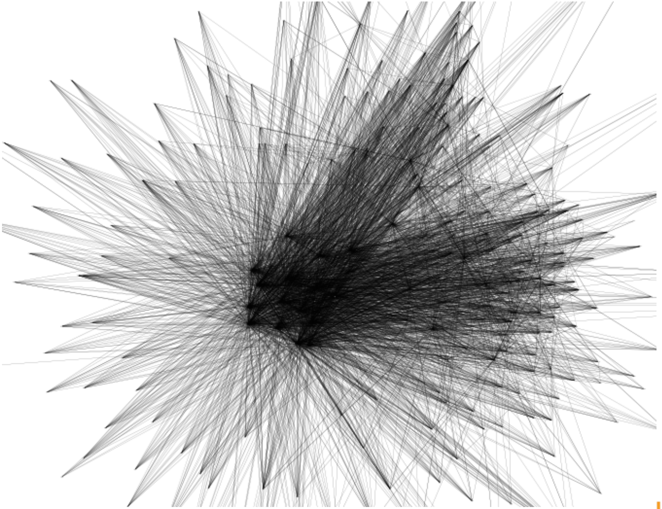
"Every flow from Secret to Internet goes through Encrypt"



Background: Assertions on flows

Not so simple on large policies

"Every flow from Secret to Internet goes through Encrypt"



Background: Assertions on flows

- Assertions are useful for expressing desired properties on information flows in security policies
- Easy to check manually for small policies
- Very hard to check manually for large policies — we need help from tools
- Useful to have assertions as
  part of the policy, and expressed in terms of the policy

Symbion

A Symbion assertion has the form

$$P \rightarrow Q : \phi$$

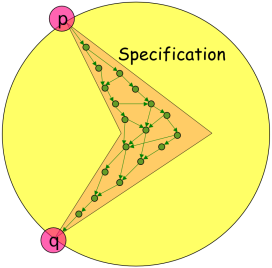
where P and Q are predicates on start and end ports, and ϕ is a predicate on what flows are acceptable between the ports. Predicates are built using regular expressions and propositional connectives.

Some examples:

- $[\mathrm{Secret}.*] \rightarrow [\mathrm{Internet}.*] : \mathrm{false}$ — *"there is no flow from any port of the Secret domain to any port of the Internet domain"*
- $[\mathrm{Secret}.*] \rightarrow [\mathrm{Internet}.*] : *[\mathrm{Encrypt}.*]*$ — *"every flow from the Secret domain to the Internet domain goes through the Encrypt domain"*

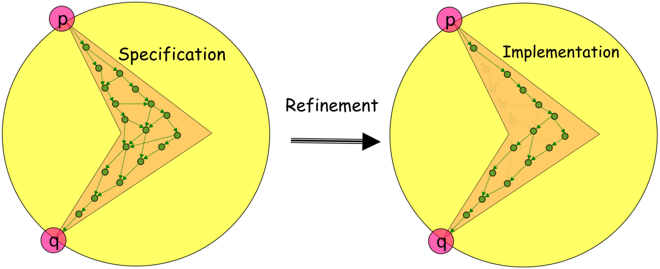
Symbion for Domain specifications

- Domain specifications can be used by a security policy designer to specify acceptable flows in domains yet to be refined
- Symbion assertions define the set of conceivable valid flows:



Symbion for Domain specifications

- When a developer refines the policy, the actual flows must be a subset of the valid flows



Symbion for Domain specifications

Example of a guard domain specification with Symbion assertions:

```
class Guard() {
  port unclassified;
  port classified;
  port output;

  assert unclassified -> output  : true;
  assert   classified -> output  : *[Declassify.*]*;
}
```

Challenges with SELinux

- How can the Reference Policy be analyzed without macro expansion? — *by using Shrimp, treating the Reference Policy as a proper domain-specific language*
- How can we understand information flow without looking inside policy modules? — *by using Lobster, making all information flows explicit*
- How can we explicitly state restrictions in information flows between modules? — *by using Symbion, expressing assertions over flows in Lobster policies*

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

Status and future

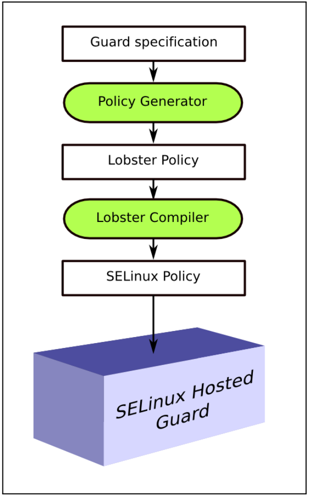
Conclusion

Experience

Project Guardol — with Rockwell Collins

Security policies from guard specifications

- The resulting policy locks down the guard components
- The Lobster policy is suitable for evaluation — high level and readable



Experience

Policies for Secure Virtual Platforms

- Lobster policies for describing information flow through event channels and grants
- Compiled into a XSM (Xen Security Module) policy

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

Symbion — Policy Properties

Experience

Status and future

Conclusion

Current status

what we have

- Lobster compiler to SELinux
- Shrimp analyzer from Reference Policy
- Design of Symbion assertion language

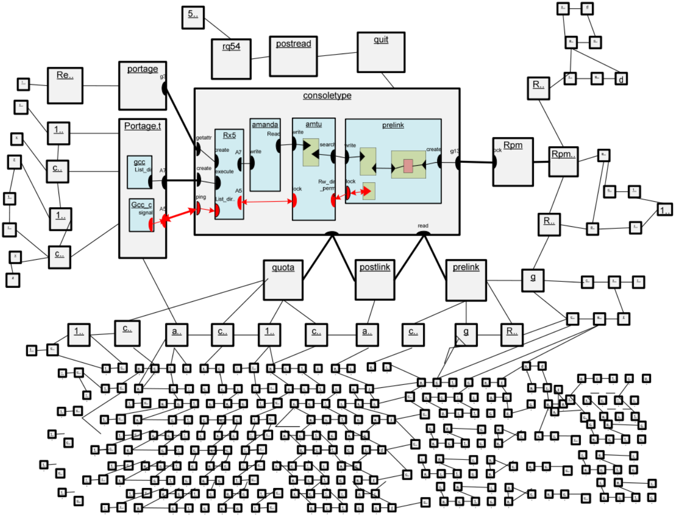
Current status

what we are working on

- Reverse compiling Reference Policy into Lobster
    - automatic discovery of domain hierarchies
    - gives us high-level information flow analysis of SELinux
- Implementing Symbion assertion checks
- Describing information flow among virtual machines
    - Lobster for designing Xen security policies
- Prototyping visualization tools

Future

Visualizing Lobster policies

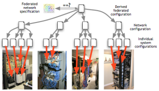


| galois |

Future

Going beyond SELinux

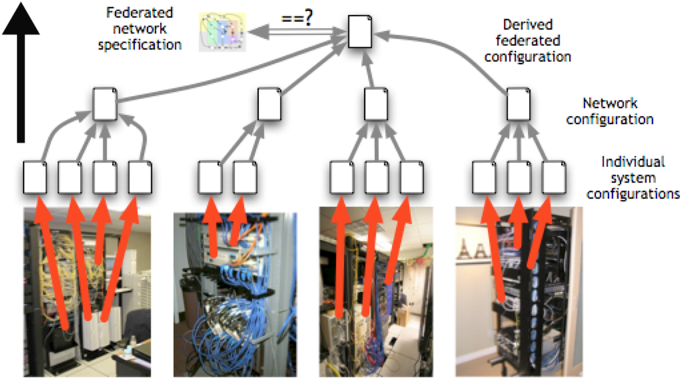
- Use Lobster to describe information flow in networks of guards, firewalls, routers, virtual machines


Federated network specification
==?
Derived federated configuration
Network configuration
Individual system configurations

- SELinux would be one of many "back-ends"
- Make tools and specifications open to invite back-end development by community
- Use Lobster to describe information flow inside programs (connect to ASA - Automated Security Assurance)

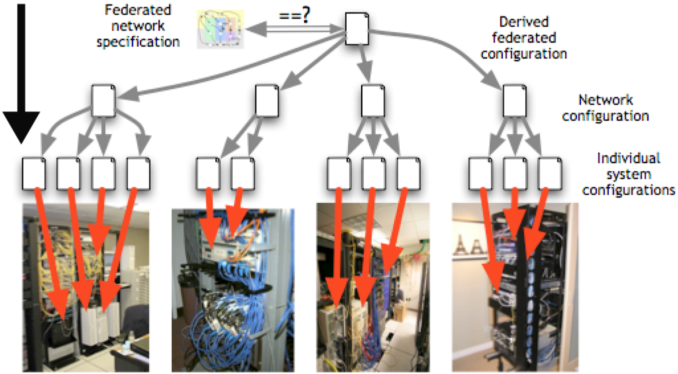
Future

Describing information flow in complex system


Federated network specification
==?
Derived federated configuration
Network configuration
Individual system configurations

Future

Prescribing information flow in complex system

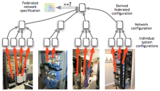


Federated network specification
==?
Derived federated configuration
Network configuration
Individual system configurations

Future

Information flow in complex systems — opportunities

- Continuous, on-line analysis of existing organization



Federated network specification
==?
Derived federated configuration
Network configuration
Individual system configurations

- Dashboard visualizes differences between prescribed and analyzed policies
- Having one comprehensive, consistent description allows us to express and check more properties, like *defense in depth*
- Extend Lobster to express *trust relationships*
- Not only machines but people can be described and assigned trust levels

Talk Plan

Background

Shrimp — SELinux Policies Made Precise

Lobster — Domains and Information Flows

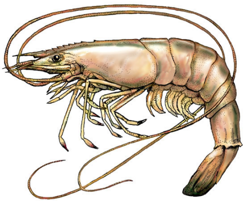
Symbion — Policy Properties

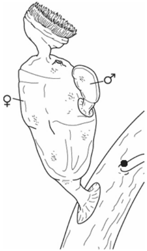
Experience

Status and future

Conclusion

Conclusion





- Shrimp — Precise understanding of complex SELinux security policies
- Lobster — High-level description of security policies in terms of nested security domains and information flows
- Symbion — High-level properties over security policies
- Future — lots of opportunities!