Symbion: An Assertion Language for Lobster Policies

Joe Hurd, Magnus Carlsson and Peter White
Galois, Inc.

{joe,magnus,peter}@galois.com

1 April 2009

Abstract

This paper defines the syntax and semantics of the Symbion assertion language for Lobster policies.

1 Introduction

The Lobster policy language is intended to model information flow in large and complex systems. Once a system has been modelled in a formal language, it is possible to check that its information flows satisfy the system security policy. This document presents two mechanisms for this: the Symbion assertion language and policy refinement.

2 Information Flows

2.1 Lobster Information Flow Graphs

Consider an information flow graph G generated by a Lobster policy. G consists of a set of nested security domains d_i, each having a set of ports $d_i.p_j$, and a set of directed connections that connect together two domain ports. Ports may be declared with information flow properties, and connections may only connect ports with compatible flow properties. Connections must not

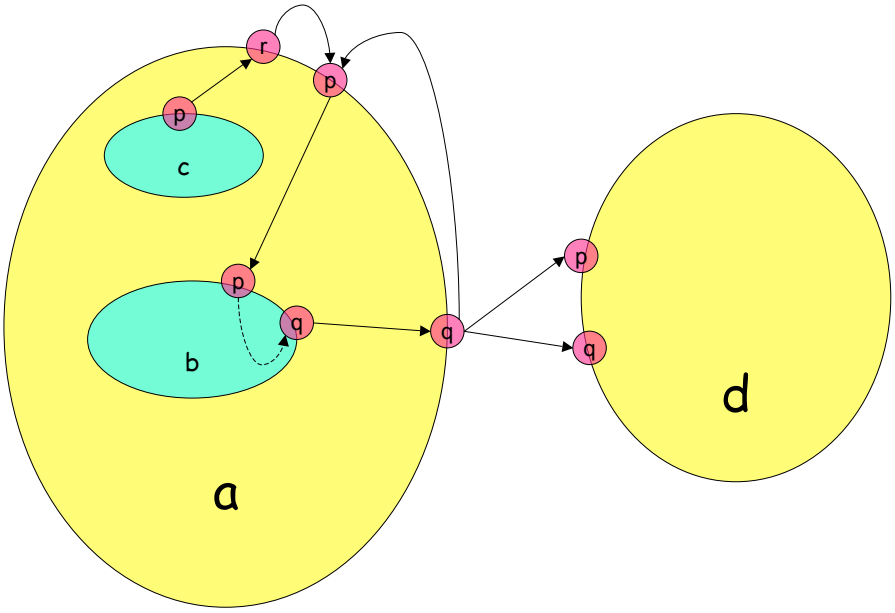

Figure 1: An example information flow graph.

cross domain boundaries, and thus for every domain d all connections are either completely inside or completely outside d. Note that it is possible to declare a connection between two ports $d.p$ and $d.q$ of the same domain d, as long as the information flow properties of the two ports are compatible, and by convention this connection is outside the domain d.

2.2 Internal Connections

In this paper we extend the Lobster concept of an information flow graph G by introducing directed *internal connections* that connect one port of a domain to another through the inside. For example, it is permissable to declare an internal connection l that connects the port $d.p$ to the port $d.q$ through the inside of the domain d. There is an important difference between internal connections and regular connections, which is that internal connections are allowed to connect ports with incompatible information flow properties.

Figure 1 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.

2.3 Flows

A *flow* in G of length n 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.

There is a natural concatenation operator that takes as arguments a flow f, a port $d.p$ and another flow g, and results in the flow $f \; d.p \; g$.

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 1 all of the following are flows:[1]

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

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

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

3 Symbion

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 programs. The idea is that flow assertions can be attached to Lobster programs, the information flow graph is then generated from the Lobster program, and the possible flows are checked to make sure that they conform with the assertions.

3.1 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

$$P \rightarrow Q : \phi$$

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.

3.2 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 [3], 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 1.

- $[a.*] \rightarrow [d.*]$: 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"*.

3.3 Checking Assertions

To demonstrate the possibility, here is one algorithm for checking that a Symbion assertion holds for an information flow graph. We do not claim that this is the most efficient way of checking assertions.

1. Compile the information flow graph to a non-deterministic finite state automaton A_G:
   (a) The ports and connections are the states.
   (b) The connections (both regular and internal) are compiled into two state transitions: the first from the source port state to the connection state; and the second from the connection state to the destination port state.

(c) All ports are possible initial states.

(d) All ports are accepting states.

This automaton accepts the precise set of flows (with start and end ports) that are possible in the information flow graph.

2. Compile the Symbion assertion to a non-deterministic finite automaton A_ϕ, using standard automata algorithms [1]. This automaton accepts the precise set of flows (with start and end ports) that will not violate the assertion.

3. Construct the automaton $A = A_G \wedge \neg A_\phi$. This automaton accepts precisely the set of flows (with start and end ports) that are possible in the information flow graph and that violate the assertion.

4. Use a standard automata algorithm to decide whether the automaton A accepts any flows [1]. If not, report success. If so, report an assertion violation and give a concrete violation, consisting of a start and end port, and a flow between them.

4 Refinement

Section 3 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.

4.1 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 $L_{p \to q}$ for every pair of domain ports p and q.

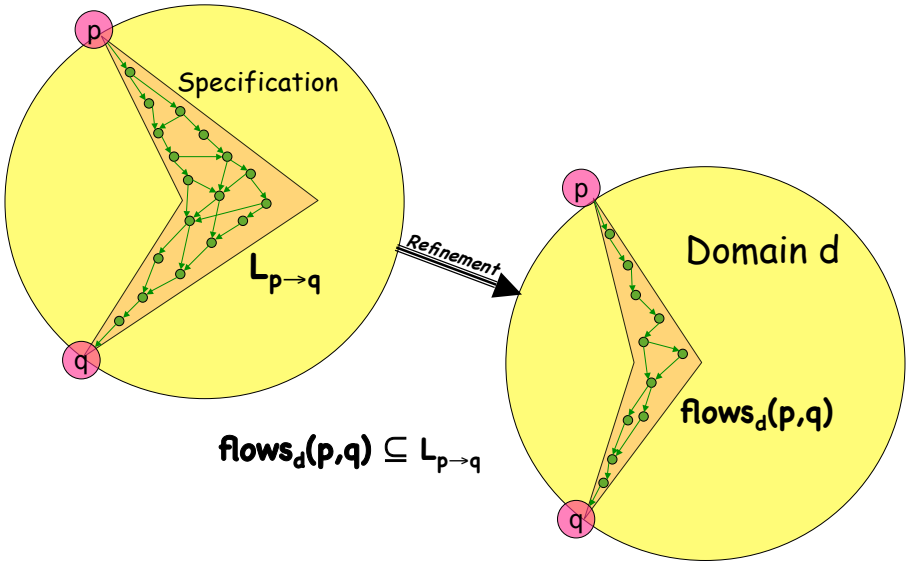

Figure 2: Verifying that the flow through the internal structure of a domain satisfies its domain specification.

4.2 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. Run the Lobster type checking algorithm [2] over the internal structure of d to infer a set of information flow properties for each port of d. Check that the specified information flow properties are more specific than the inferred properties, for each port.

3. For each pair of ports p and q, compute the set of flows $\text{flows}_d(p, q)$ from p to q through the internal structure of d. Check that the computed set of flows satisfies the Symbion flow predicate $L_{p \to q}$, by using the assertion checking algorithm in Section 3.3. This step is illustrated in Figure 2.

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.

4.3 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.

4.4 Primitive Domains

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

References

[1] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2nd edition, 2001.

[2] Joe Hurd, Magnus Carlsson, Brett Letner, and Peter White. Lobster: A domain specific language for selinux policies. Galois internal report, December 2008.

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