SINS: A Middleware for Autonomous Agents and Secure Code Mobility

In Proc. *Security of Mobile Multiagent Systems (SEMAS-2002).* AAMAS 2002, Bologna, Italy. July 16, 2002

[Extended Abstract]

Ramesh Bharadwaj
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC, 20375-5320 USA
ramesh@itd.nrl.navy.mil

1. INTRODUCTION

Building trusted applications is hard, especially in a distributed or mobile setting. Existing methods and tools are inadequate to deal with the multitude of challenges posed by distributed application development. The problem is exacerbated in a hostile environment such as the Internet where, in addition, applications are vulnerable to malicious attacks. It is widely acknowledged that intelligent software agents provide the right paradigm for developing agile, re-configurable, and efficient distributed applications. Distributed processing in general carries with it risks such as denial of service, Trojan horses, information leaks, and malicious code. Agent technology, by introducing autonomy and code mobility, may exacerbate some of these problems. In particular, a malicious agent could do serious damage to an unprotected host, and malicious hosts could damage agents or corrupt agent data.

Secure Infrastructure for Networked Systems (SINS) being developed at the Naval Research Laboratory is a middleware for secure agents intended to provide the required degree of trust for mobile agents, in addition to ensuring their compliance with a set of enforceable security policies. An infrastructure such as SINS is central to the successful deployment and transfer of distributed agent technology to Industry because security is a necessary prerequisite for distributed computing.

2. SECURITY REQUIREMENTS OF MOBILE AGENTS

The following requirements of secure mobile agents (see [5]) are addressed by SINS:

- The author and initiator of an agent must be authenticated.
- The integrity of an agent's code must be checked.
- Interpreters must ensure that agent privacy is maintained during data exchange.
- Interpreters must protect themselves against malicious agents.
- Interpreters must ensure that migrating agents are in a safe state.
- Agents must protect themselves from malicious hosts and interpreters.
- An initiator must be able to control an agent's flexibility; i.e., restrict or increase an agent's authorization in specific situations.
- Initiators must be able to control which interpreters are allowed to execute their agents.

3. SINS ARCHITECTURE

Figure 1 shows the architecture of SINS. Agents are created in a special purpose synchronous programming language called Secure Operations Language (SOL) [1]. A SOL application comprises a set of modules, each of which runs on an Agent Interpreter (AI). The AI executes the module on a given host in compliance with a set of locally enforced security policies. A SOL application may run on one or more AIs, spanning multiple hosts across multiple administrative domains. Agents are created using a visual language known as visual SOL (vSOL) in an Agent Creation Environment (ACE), and are automatically translated into SOL. Agent Interpreters communicate among themselves using an inter-agent protocol [7], similar to SOAP/XML [8].

Currently, protection of agents from malicious hosts is an area of active research. Therefore, in our initial implementation of SINS, we assume a degree of trust among the hosts. This is reasonable, especially in a large organization such as the Department of Defense, where one may assume that extant policing methods and techniques for intrusion detection are able to identify and isolate malicious hosts and eavesdroppers. We plan to address the more general problem

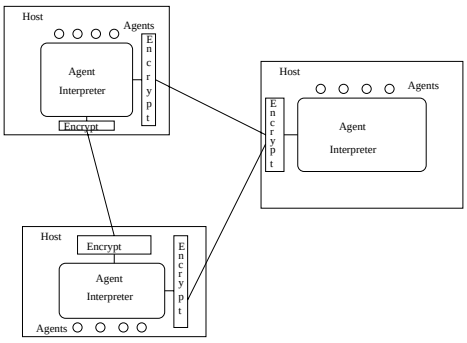

Figure 1: Architecture of SINS.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2002		**00-00-2002 to 00-00-2002**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
SINS: A Middleware for Autonomous Agents and Secure Code Mobility	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**3**	

of survivability and agent protection in our future work. Therefore, the current SINS implementation assumes the following:

- A host will run an agent interpreter to completion.
- All agent interpreters will run agents correctly.
- An agent interpreter will transfer agent data as requested.
- Agents' code and data cannot be kept private from hosts.
- Agent-to-agent communication cannot be kept private from hosts.
- Agents cannot carry secret keys.

4. TECHNICAL APPROACH

The SINS middleware and the associated Agent Creation Environment are designed to explicitly address requirements of security and high assurance described above, in addition to related problems of agent creation, deployment, and integration into the host after deployment. Although security is our primary concern, we also address problems of efficiency, reconfigurability, and ease of agent creation and debugging. SINS addresses each of the security requirements as detailed below:

4.1 Authentication and Authorization

In SINS, code distribution is distinct from agent instantiation. Consequently, the issue of code tampering by possibly compromised hosts is addressed. Agent code resides in a SOL code repository and is digitally signed. Hosts retrieve the code either directly from the repository or from another host that has the most recent cached copy. The execution of a set of agents comprising a SOL application is initiated by a single host. SINS provides role-based access control & management (RBAC) and trust management (TM) to authenticate the agent initiator and to provide access to resources on a given host. The initiating host has fine-grained control over each agent in the application. This gives the initiator the ability to run the agents with restricted authority in most cases, but with greater authority in certain situations.

4.2 Integrity of Agent Code

All agents are programmed in SOL, a *verifiable* synchronous language [1]. As opposed to agents developed in a Turing-complete general purpose language such as Java, whose properties such as termination are undecidable, many properties of agents programmed in SOL, a more restricted language, are decidable. All analyzed and verified SOL agents are guaranteed to have no unbounded loops, violations of array index bounds, buffer overflows, etc. Therefore, as opposed to other agent systems where code integrity is based merely on trusting the author of the agent's code, integrity of agent code is ensured in SINS by proving (with mathematical certainty) the safety of SOL agents and their compliance with a host's local security policies. SINS includes a compliance checker (CC) [3] which establishes formally the compliance of the behavior exhibited by a SOL agent, or a set of agents, with the required security properties and the local security policies.

4.3 Agent Privacy

Agent Interpreters communicate among themselves using a secure protocol (SSL) which ensures agent privacy during data exchange and prevents casual intruders from eavesdropping on inter-agent message exchanges. Also, SINS implements a security architecture for monitoring and coordinating agents' activities.

4.4 Protection from Malicious Agents

Since SOL agents are composable and modular, CC can evaluate emergent behavior of agent communities, which is generally not possible in the absence of an agent aggregation framework. This capability enables early detection and prevention of an organized, cooperative attack in an environment in which each agent performs some action that falls beneath the threshold of most analysis techniques, but effects serious damage as a distributed attack. Currently these types of vulnerabilities have defied formal analysis.

4.5 Safe Agent Migration

Because a migrating agent can become malicious, we equip each agent in SINS with an appropriate state appraisal function which is used each time an interpreter activates an agent. The state appraisal function ensures that an agent will perform as required and that its data has not been tampered with. The static analysis tool CC can guarantee that the state appraisal function satisfies key safety properties and is in conformance with security policies being enforced at a given host.

4.6 Agent Protection from Malicious Hosts

As we mentioned before, SINS agents are currently not fully protected from a malicious host. However, the likelihood of agent corruption by a host is minimized by the introduction of a special class of agents called security agents [2] that police other agents such as application agents developed to support a given distributed application. Security agents protect a system against Information Operation (IO) attacks by implementing key security features such as encryption, authorization, policy enforcement, virus checking, and intrusion detection. Since security agents have more privileges than application agents, we need higher assurance during their development, deployment, and integration that they are safe and secure. This is achieved by the three-pronged approach of programming them in a safe language (SOL), by applying the compliance checker to establish formally their compliance with required safety properties, and by the security architecture for monitoring and coordinating agents' activities.

5. SECURITY AGENTS

In this section, we shall examine how *enforceable* safety and security policies [6] are expressed as *Security Agents* in SOL. The enforcement mechanism of SOL works by terminating all instances of an agent for which the safety or security policy being enforced no longer holds.

5.1 A Brief Introduction to SOL

A module is the unit of specification in SOL and comprises variable declarations, assumptions and guarantees, and definitions. The `assumptions` section typically includes assumptions about the environment of the agent. Execution aborts

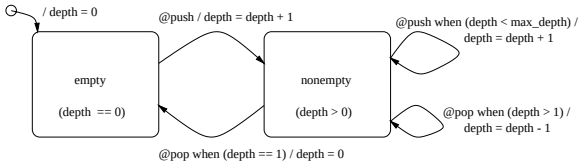

Figure 2: vSOL representation of `safestack`.

when any of these assumptions are violated by the environment. The required safety properties of an agent are specified in the `guarantees` section. The `definitions` section specifies updates to internal and controlled variables.

A variable definition is either a *one-state* or a *two-state* definition. A one-state definition, of the form $x = expr$ (where *expr* is an expression), defines the value of variable `x` in terms of the values of other variables *in the same state*. A two-state variable definition, of the form $x =$ `initially` *init* `then` *expr* (where *expr* is a two-state expression), requires the initial value of `x` to equal expression *init*; the value of `x` in each subsequent state is determined in terms of the values of variables in that state *as well as the previous state* (specified using operator `PREV` or by a `when` clause). A *conditional expression*, consisting of a sequence of branches "`[]` guard $\rightarrow$ expression", is introduced by the keyword "`if`" and enclosed in braces ("`{`" and "`}`"). A guard is a boolean expression. The semantics of the conditional expression `if` { $[]g_1 \rightarrow \text{expr}_1$ $[]g_2 \rightarrow \text{expr}_2 \ldots$ } is defined along the lines of Dijkstra's *guarded commands* [4] – in a given state, its value is equivalent to expression expr_i whose associated guard g_i is true. If more than one guard is true, the expression is nondeterministic. It is an error if none of the guards evaluates to `true`, and execution aborts. The *case expression* `case` *expr* { $[]v_1 \rightarrow \text{expr}_1$ $[]v_2 \rightarrow \text{expr}_2 \ldots$ } is equivalent to the conditional expression `if` { $[](expr == v_1) \rightarrow \text{expr}_1$ $[](expr == v_2) \rightarrow \text{expr}_2 \ldots$ }. The conditional expression and the case expression may optionally have an `otherwise` clause with the obvious meaning.

5.2 Safety Property Enforcement

We examine how SOL Security Agents are used to enforce safety properties. The example we shall use is a stack, which has the associated methods `push`, `pop`, and `top`. Informally, `push(x)` pushes the value of integer variable `x` on the stack and `pop()` pops the topmost value off the stack. The method `top()` returns the current value at the top of the stack and leaves the stack unchanged. The stack can accommodate at most `max_depth` items. The safety policies we wish to enforce are: (i) No more than `max_depth` items are pushed on the stack. (ii) Invocations of methods `top` and `pop` are disallowed on an empty stack. Figure 3 shows a SOL module `safestack` which enforces these safety policies on *all* SOL modules which use the stack object (implemented in the embedding language). Figure 2 is the module `safestack` rendered in the visual syntax of SOL using ACE. Note that by deliberately omitting the `otherwise` clauses in the `if` statements, we abort the execution of an agent when none of the guards is `true` during execution. If this is too drastic, corrective action may be specified in an `otherwise` clause; for example, to ignore all `push` actions when the stack is full.

6. CONCLUSIONS

The goal of the NRL secure agents project is to develop enabling technology that will provide the necessary security infrastructure to deploy and protect time- and mission-critical applications on a distributed computing platform, especially in a hostile computing environment such as the Internet. Our intention is to create a robust and survivable information grid that will be capable of resisting threats and surviving attacks. One of the criteria on which this technology will be judged is that critical information is conveyed to principals in a manner that is secure, safe, timely, and reliable. No malicious agencies or other threats will be able to compromise the integrity or timeliness of delivery of this information.

```
deterministic reactive module
  safestack(integer max_depth) {
interfaces
  void    push(integer x);
  void    pop();
  integer top();
internal variables
  {empty, nonempty} status;
  integer in [0:max_depth] depth;
guarantees
 INV1 =
  (status == empty) <=> (depth == 0);
definitions
  [status, depth] = initially [empty, 0] then
  case PREV(status) {
    [] empty ->
      if {
        [] @push -> [nonempty, PREV(depth) + 1]
        // other operations illegal!
      }
    [] nonempty ->
      if {
        [] @top ->
           [PREV(status), PREV(depth)]
        [] @pop when (depth > 1)  ->
           [nonempty, PREV(depth) - 1]
        [] @pop when (depth == 1) ->
           [empty, 0]
        [] @push when (depth<max_depth) ->
           [nonempty, PREV(depth) + 1]
        // @push when (depth == max_depth) illegal!
      }
  }; // end case
} // end module safestack
```

Figure 3: Security agent for safestack.

7. REFERENCES

[1] R. Bharadwaj. SOL: A verifiable synchronous language for reactive systems. In *Proc. Synchr. Languages, Apps., and Programming, ETAPS 2002*, Grenoble, France, April 2002.

[2] R. Bharadwaj et al. An infrastructure for secure interoperability of agents. In *Proc. Sixth World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, Florida, July 2002.

[3] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. 6^{th} TACAS, ETAPS 2000*, Berlin, March 2000.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[5] W. M. Farmer et al. Security for mobile agents: Issues and requirements. In *Proc. National Information Systems Security Conference*, October 1996.

[6] F. B. Schneider. Enforceable security policies. *ACM Trans. Infor. and System Security*, 3(1):30–50, February 2000.

[7] E. Tressler. Inter-agent protocol for distributed SOL processing. Technical Report To Appear, Naval Research Laboratory, Washington, DC, 2002.

[8] W3C. Simple Object Access Protocol (SOAP) 1.1. Technical Report W3C Note 08, The World Wide Web Consortium, May 2000.