MASSA:
Mobile Agents Security through Static/Dynamic Analysis

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
vigna@cs.ucsb.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
harrold@cc.gatech.edu

Abstract

Existing Mobile Agent Systems (MASs) suffer from security problems that must be solved, if mobile code is to be used in the development of mission-critical, real-world applications. In this paper we propose a framework that provides automated support for verification and analysis of MASs and allows for identifying security issues before the MASs are placed into action. The proposed approach is centered around an abstract reference model and combines static and dynamic security analysis techniques explicitly tailored to mobile agent systems. Our preliminary work shows that, in the cases we studied, appropriate analysis techniques facilitate the identification of security vulnerabilities in MASs.

Keywords: Mobile Agent Systems, Security Analysis, Static Analysis.

1 INTRODUCTION

Mobile-code technologies [4, 7] are attracting a great deal of interest from both industry and academia. In particular, the mobile-agent paradigm has been used successfully to design applications ranging from distributed information retrieval [8], to network management [2], to wireless-based services for dynamically reconfigured network links [14].

Although the mobile-agent paradigm provides a number of advantages with respect to the traditional client-server approach, some fundamental issues must be addressed for this new approach to be universally accepted. In particular, security and verification issues cause major concerns among potential users of mobile agent systems. *Mobile Agent Systems* (MASs) provide a distributed computing infrastructure composed of *places* where agent-based applications belonging to different (usually untrusted) users can execute concurrently. The places that compose the infrastructure may be managed by different authorities (e.g., a university or a company) with different and possibly conflicting goals, policies, and security requirements, and may communicate across untrusted communication infrastructures, such as the Internet.

Even though the security issues related to the use of MASs have been studied in different contexts by both the distributed-systems and the computer-security communities, the results achieved in these fields have seldom been used either to secure existing MASs or to design and implement secure new MASs. Unfortunately, MASs are often proof-of-concept prototypes whose focus is on mobility mechanisms, and security is left as future work. Some existing systems do provide some basic mechanisms for security and for the definition of security policies, but these mechanisms are usually primitive. Moreover, the security mechanisms available today are far from providing a sound, comprehensive security solution—the results achieved to date address only a subset of problems and provide only specific solutions for such problems (e.g., [9, 15]).

In short, existing MASs suffer from security problems. These problems must be overcome if mobile code is to be used in the development of mission-critical, real-world applications. To solve these issues, we need a general solution that, independent of the particular mobile-agent technology under consideration, is able to guide and support MAS developers in the analysis of security issues during the design and implementation of services and applications.

In this paper, we propose a framework that provides automated support for modeling and analysis of security issues in MASs, and allows for an early verification of MASs before they are placed into action. The framework is based on a novel approach that integrates static and dynamic analyses so that the results from the former can be used to guide the latter and vice versa. This integration is achieved through the presence of an overall abstract reference model.

Our preliminary results show that, in the cases we studied, appropriate analysis techniques facilitate identification of security vulnerabilities in MASs. In particular, we have successfully applied dynamic, black-box techniques to identify threats to the security of MASs, and static, white-box techniques to diagnose early the possible presence of such threats. We are currently performing further research to identify additional types of analyses that may help evaluating the security of existing MASs and thus provide guidance in developing new secure MASs. We strongly believe that the complementary use of static and dynamic analysis techniques is the key to addressing many security problems in a general and efficient way.

2 A FRAMEWORK FOR EVALUATING THE SECURITY OF MOBILE AGENT SYSTEMS

Security vulnerabilities in MASs can be exploited in different ways by a determined attacker (see Reference [6] for a list of examples). Unfortunately, these kinds of vulnerabilities are not visible and obvious, but rather hidden in the peculiarities of the system. MASs are complex, and we believe that they must be verified through dynamic or static analysis after the system is developed and before it is deployed and used. This approach reflects the approach that we usually follow when developing software systems in general: although we have powerful design techniques and programming languages enforcing good programming practices, verification of the resulting software is always necessary.

Our overall goal is to define a general approach for the verification of MASs that is able to provide both the developer and the user of a MAS with a higher level of confidence in the security mechanisms provided by the system. We are convinced that such assurance is a necessary prerequisite for the widespread acceptance of applications based on mobile agents. To this end, we propose a framework that is composed of a reference model and a set of static and dynamic analysis techniques. The reference model lets us develop a general approach to the problem of MAS security by abstracting security mechanisms and by defining attack classes in a technology-independent way. The set of dynamic

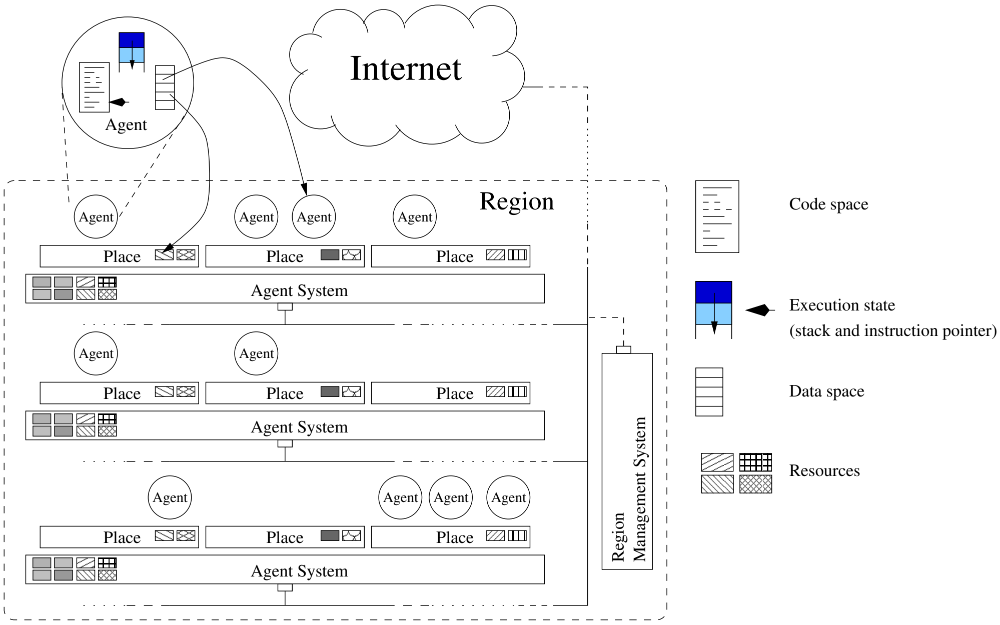

Figure 1. An abstract reference model for MASs.

techniques consists of a library of attack patterns defined on the reference model. The attack patterns focus on the authentication, authorization, and accounting functionalities provided by MASs. The attack patterns can be instantiated into actual agents that perform particular tests on the system under analysis. The set of static techniques consists of program analysis techniques that are (1) existing techniques that we adapt or extend, (2) novel approaches that we develop, or (3) combinations of techniques that we integrate. We will use several such techniques to perform analysis of the code of the MASs, including exception analysis, escape analysis, and data-dependence analysis (as discussed in Section 2.2).

We exploit the framework to verify whether a given MAS is secure using the following approach: (1) map the MAS onto the reference model; (2) based on the mapping between the model and the MAS, identify a set of security issues for the system; (3) analyze the involved security mechanisms of the MAS using static and dynamic analysis techniques; and (4) if necessary, extend the framework to account for new characteristics of MASs or new classes of threats to their security not yet considered. Beside its generality, one major advantage of the framework is that it facilitates automation. Some steps, such as 2 and 3, can be fully automated, and the rest can be supported by suitably defined tools.

2.1 A Reference Model for Security Analysis

As discussed in the Introduction, existing approaches for assessing the security of MASs lack generality and in many cases address only specific problems. To provide a general approach to the problem of MAS security, we first define a reference model that can help in abstracting security mechanisms and in defining attack classes in a way that is independent of a specific technology. In this way, we can factor the knowledge gathered through the analysis of different systems, and leverage previous experience in evaluating the security of MASs. The definition of an abstract reference model has several advantages: (1) it provides common concepts, abstractions, and terminology for the security analysis; (2) it allows for highlighting the security *abstractions* available in the different languages; and (3) it supports the definition of general attack classes that can then be instantiated on particular systems.

In previous work, Fischmeister, Vigna, and Kemmerer proposed a preliminary reference model (shown in Figure 1), defined based on the analysis of a number of existing systems and their security models. The main components of the model are mobile agents, places, agent systems, regions, and principals. *Mobile agents* are computational units consisting of a code space, an execution state, and a data space. The code space contains a set of references to code fragments that can be invoked during the execution of an agent. The execution state contains all the information related to the evolution of an agent and the program counter. The data space contains references to external resources that can be accessed by an agent. *Places* support the execution of agents. Each place provides a local infrastructure to a visiting mobile agent. The place infrastructure supports the execution of particular procedures as defined in the associated code repository and provides access to local resources. Access to local resources is regulated by the place's security system, which is composed of three subsystems whose tasks are authentication, authorization, and accounting. Each subsystem contains a policy that specifies how the security functionality is configured, a set of security resources that represent dynamic information about the state of the system, and a code repository that contains the definition of the procedures used to implement the security subsystem mechanisms. Due to space reasons, the model is described in detail elsewhere [6].

The model has been successfully used to describe the security mechanisms of four MASs, namely, Grasshopper,[1] Agent TCL,[2] Aglet,[3] and Jumping Beans.[4] We will extend this model and use it to analyze additional MASs. To this end, we will first select a set of additional MASs to be considered, and then fit each of the considered MASs into the reference model, to verify whether the model is adequate to represent the characteristics of the MAS or it needs to be extended. We will also extend the model by including other reference models, such as the OMG's MASIF specification [1].

2.2 Security Analysis Techniques for MASs

We use the reference model as a basis for identifying a set of security threats (i.e., possible attack patterns) and analysis techniques to address them. For example, in the case of authorization mechanisms, the security threats can be identified by means of an *access matrix*. Intuitively, the access matrix helps to identify the possible *access space* for a component, that is, which other components in the model can be accessed (e.g., by means of an object reference or a file descriptor). The analysis of a system is performed by analyzing the different access matrices and populating the types of access allowed between components implemented in the particular system. For each possible access, the possible operations and subset of these operations that would actually be permitted must be determined. Then each operation is exercised and the outcome is verified against the defined policy.

Once the possible threats to security are identified on the model, the attacks are mapped onto the particular MAS under analysis. This provides an instantiation of the attack pattern for dynamic analysis and defines the element of the MAS implementation to be verified using static analysis techniques. As an example, consider two attack patterns: an attack pattern aimed at the disclosure of a MAS code repository and an attack pattern whose goal is to exercise the access to the MAS policy database. The code disclosure attack pattern can be informally expressed as follows: (1) raising of uncaught exceptions; (2) inspection of the stack to collect class names; (3) use of reflection classes to gather information on the code repository through the class names previously identified; and (4) identification of static methods in the code repository. The policy database attack pattern is "triggered" by the fact that a reference to the class implementing the policy can be obtained by calling one of the static methods identified during the previous attack. The policy database attack pattern can be expressed as follows: (1) obtain a reference to the policy component; (2) exercise read access on all the elements of the component API; and (3) exercise write access on all the elements of the component API. These two attack patterns were instantiated in the context of the Aglet system [10], and led to a compromise. The attacks let us spot the specific chain of problems causing the vulnerability: (1) the "tricky" interaction between the agent and the MAS was not anticipated by the designers of the Aglet security system (i.e., an aglet was not supposed to access the policy database); (2) as a consequence, modifications to the policy database were not thoroughly checked by the Security Manager.

The analysis of this vulnerability provides input and scope for the corresponding, complementary static analysis. For this vulnerability, we have identified three static analysis techniques that are suitable: exception analysis, escape analysis, and data-dependence analysis. In previous work, Harrold and Sinha developed techniques to represent exception-handling constructs [16]; Orso, Harrold, and Sinha studied data dependences in complex languages [12, 13]; and Harrold and Liang tailored existing escape analysis techniques [3, 5] to the Java language [11]. To address this vulnerability, we combined those techniques (1) to enable the detection of exceptions that can be raised within a system and are not caught by any exception handler, and (2) to consequently identify a set of methods and attributes that are visible through analysis of stack traces in such uncaught exceptions. The instantiation of this analysis

[1] `http://www.ikv.de/products/grasshopper/`
[2] `http://www.cs.dartmouth.edu/~agent`
[3] `http://www.trl.ibm.co.jp/aglets/`
[4] `http://www.jumpingbeans.com`

for the Aglet system provided us with a list of possibly accessible methods and attributes that were not part of the Aglet's published API. This information can be provided to the MAS developer as a warning about possible security problems, or can be used to perform further analyses, both static and dynamic.

This example illustrates the general approach that we propose. The approach, as we stated above, is to use dynamic techniques to identify security vulnerabilities, and then develop static analysis techniques that are appropriate for the identification of the problem(s) causing such vulnerabilities. By doing this, we can populate the framework with a rich set of analysis techniques that address different security issues.

The techniques that we will develop depend on the security vulnerabilities we find during our dynamic analyses. We will build on our previous work in analysis, and adapt, extend, develop, or combine techniques required for detecting security threats. The main problem with this use of static analysis techniques is that, due to their computational cost in terms of both space and time, practical techniques usually compute conservative approximations of the results. This approximations can lead to the reporting of spurious results when evaluating the security of a MAS. For example, the analysis of the Aglet system identified several public static methods, but the method used to access the Aglet policy database may be the only method actually "dangerous" for the system.

We account for this problem in two ways. First, we adapt and extend the static analysis techniques. More precisely, we can adapt an analysis by tuning its precision (to decrease the number of spurious results) or extend the analysis technique by including different kinds of analyses (to "filter" the results and thus increase their accuracy). For example, the addition of an analysis of the Security Manager, to verify which accesses to the system are checked and which ones are not, may eliminate some of the methods identified through escape analysis. Second, we use static and dynamic analyses jointly, when the results of the former are too imprecise to be useful. In this second case, the results of the static analyses will be still useful to avoid performing part of the dynamic analyses, so as to increase the efficiency of the overall analysis—in general, the static analysis techniques that we consider are safe, and therefore their results can be used to guide the dynamic analysis by pruning the analysis space.

References

[1] MASIF - Mobile Agent System Interoperability Facility. Draft, Oct. 3 1998.

[2] M. Baldi and G. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In R. Kemmerer, editor, *Proc. of the 20th Int. Conf. on Software Engineering*, 1998. To appear.

[3] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 20–34, Oct. 1999.

[4] A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 1–19, Oct. 1999.

[6] S. Fischmeister, G. Vigna, and R. Kemmerer. Evaluating The Security Of Mobile Agent Systems. Technical report, University of California, Santa Barbara, 2000.

[7] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[8] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In K. Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents: 1st International Workshop MA '97*, volume 1219 of *LNCS*, pages 39–49. Springer, Apr. 1997.

[9] F. Hohl. Time Limited Blackbox Security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.

[10] D. Lange and D. Chang. IBM Aglets Workbench—Programming Mobile Agents in Java. IBM Corp. White Paper, Sept. 1996.

[11] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analysis for java. Submitted for publication, February 2001.

[12] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. In *Proc. of the 9th International Workshop on Program Comprehension*, May 2001. (To appear).

[13] A. Orso, S. Sinha, and M. J. Harrold. Incremental slicing based on data-dependences types. Submitted for publication, January 2001.

[14] G.-C. Roman, G. P. Picco, and A. L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.

[15] T. Sander and C. Tschudin. Protecting Mobile Agents Againts Malicious Hosts. In *Mobile Agents and Security*, volume 1419. Springer-Verlag, 1998. ISBN 3-540-64792-9.

[16] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26(9,):849–871, September 2000.