Security Issues in Mobile Code Paradigms

Simão Melo de Sousa
University of Beira Interior, Portugal

Mário M. Freire
University of Beira Interior, Portugal

Rui C. Cardoso
University of Beira Interior, Portugal

INTRODUCTION

Unlike mobile computing, in which hardware moves, *mobile code* moves from nodes to other nodes and can change the machines where it is executed. A paradigmatic example of such *mobile code* are Java applets that can be downloaded from a distant machine and executed by a virtual machine embedded in a browser. Multi-application smart cards (like Javacards) are an example of an execution environment that allows the loading and the execution of (mobile) programs into a card after its issuance. Code mobility allows the software reconfiguration without delivering a physical support, as done by Sun initially with Java to reprogram cable TV boxes, or nowadays, by Microsoft to promptly distribute software patches. PostScript files are another type of mobile programs which execute in printers to produce graphic images. Mobile code may also be used in distributed systems to adapt autonomously in order to balance loads or compensate for hardware failures (Brooks, 2004). Mobile code has received a great deal of interest as a promising solution to increase system flexibility, scalability, and reliability. However, to reach such objectives, some issues need to be matured, namely security issues. This article addresses security issues in *mobile code* paradigms.

BACKGROUND

Several *mobile code* paradigms have been reported (Brooks, 2004; Brooks & Orr, 2002; Fuggetta, Picco, & Vigna, 1998; Milojicic, Douglis, & Wheeler, 1999; Tennenhouse, Smith, Sincoskie, Wetherall, & Minden, 1997; Wu, Agrawal, & Abbadi, 1999). These paradigms differ on where code is executed and who determines when mobility occurs (Brooks & Orr, 2002; Brooks, 2004) and can be classified as follows:

- **Client-Server:** The user node invokes code resident on a distant node: the server or program node. This node fetches the required data from data nodes, executes the invoked program, and returns the result to the user node. Examples include the common object request broker architecture. CORBA integrates remote procedure calls (RPCs) with the object-oriented paradigm.
- **Remote Evaluation:** The user node requests the execution of code resident on a distant node. This node uploads the code to the node containing the data needed for its execution. The execution takes place in this node, and the result is then sent to the user node. Examples include CORBA, Simple Object Access Protocol (SOAP) and Web Services.
- **Code-On-Demand:** The user node requests the execution of code resident on a distant node. This code is downloaded on user node and locally executed. Examples include Java applets and Active X programs.
- **Process Migration:** The operating system dispatches processes from one node to others nodes in order to balance the load. Examples include Mosix and Sprite.
- **Mobile Agents:** The user node executes a program, called agent, which moves, along with its execution context, from node to node. The decision to move from one node to another node or to execute a specific set of operations on a particular node is made by the agent itself. The result of the execution is, at the end, transmitted within the program to the user node. There are several agent and multi-agent platforms.
- **Active Networks:** In this paradigm, the network configuration and infrastructure can be modified by the transmitted packets. Here, the packets act as mobile code. An example would be Capsules.

A mobile agent is a program that encapsulates code, data, and execution context. The mobile agent is sent by the client to another node. Unlike a procedure call, the agent does not have to return data to the client. The agent can migrate to other node, send information to the client, or come back to the client. However, the efficiency of each approach depends on network configuration and the size of programs and data files.

Copyright © 2009, IGI Global, distributing in print or electronic forms without written permission of IGI Global is prohibited.

SECURITY ISSUES IN MOBILE CODE

One of the major challenges in the context of mobile code is the safety of the execution of untrusted code. This concern occurs naturally when we verify that mobile code to be executed comes from an eventually unknown source, or it was designed or compiled by unknown methods. In fact, the code may have been produced or changed by malicious sources. Thus, an execution environment for mobile code must be able to execute mobile code without allowing it to produce damages in the case of being a malicious code.

From a theoretical point of view, the problem of stating if a given program is inoffensive or malicious is not decidable in general. Thus, the quest of finding a universal filter that rejects every malicious code and accepts innocuous programs is an utopia. It is indeed very hard to universally and formally define what is a malicious program is. However, there exist several partial solutions which increase the safety of execution environments. They can be classified in these four approaches (Rubin & Geer, 1998; Zachary, 2003):

- Sandboxes, which limit or control the context in which code is executed;
- Code signing, which ensures that code comes from a trusted source and its integrity;
- Firewalls, which limits the accessibility; and
- Proof-carrying code (PCC), in which code carries explicit proof of its safety.

The first approach consists in the isolation of the code execution zone. Each mobile program is executed within a controlled context and isolated from the other processes (including memory). Control is assured by runtime monitoring of the performed operations. For instance, sensitive operations (whether operation on resources such as disks, memory, etc., or operations such as communications or data/files handling) may be forbidden or, at least, supervised. Enforcing security policies by confinement and runtime access control is relatively easy to implement (when compared with other approaches), easy to use, and provides a reasonable level of confidence. A successful example of such approach is the Java virtual machine and its security manager mechanism. However, runtime checking induces a penalty in terms of execution performance. In the same vein, access control policies limit the computational ability of mobile code (for instance, an innocuous applet could have access to the whole instruction set).

The next approach, the *code signing* approach, allows the execution of code which presents enough credentials. This mechanism is based on the extraction and the verification of a digital signature which is included in the code to be executed. This signature allows the identification of the code producer and the code integrity. If code comes from a source identified as secure and if the code has not been changed since it had to leave the source, then the execution environment may allow its execution. Such a mechanism takes place before the execution stage. Unfortunately, it does not provide information about the actions performed by the program and must be associated with other security mechanisms. Therefore, most popular mobile code execution/support systems such as Java and .NET integrate a combination of the two approaches, since this increases the flexibility of policy securities.

Another way to guarantee the security in a mobile context is based on the restriction and control of the mobility or the communication capability. These mechanisms rely on *firewalls* and other similar mechanisms. This approach allows precise control of the generated interactions by the executed program. However, since this mechanism acts in runtime, it leads to performance degradation of program execution and of the infrastructure that supports the execution. Another drawback is that the safety cannot be fulfilled exclusively in terms of safe interaction, but this approach can be used in conjunction to other security mechanisms.

Recent and emerging approaches try to minimize the need of runtime verification. Such approaches are known as proof-carrying code (Appel, 2001; Appel & Felty, 2001;Barthe, Grégoire, Kunz, & Rezk, 2006; Colby, Lee, Necula, Blau, Plesko, & Cline, 2000; Hamid, Shao, Trifonov, Monnier, & Ni, 2002) or static program analysis. These mechanisms operate on the code as soon as it is received and can get conclusions about the safety of the program without requiring its execution. From the code consumer point of view, the penalty is located in the loading time. The underlying principle is the following: the code to be executed is enriched in such a way that it contains enough information for the execution environment to verify the conformance of the program with respect to the security policies of the code consumer. If the program is approved, then it can be executed in a safe way and these policies do not need to be verified at runtime. The several approaches in these families of mechanisms differ in the quantity of the information required in the code to be executed. This information can vary from complete demonstrations (as the name "proof-carrying code" suggests) to simple type annotations. For instance, Java bytecode, the code executed by the Java virtual machine, is a typed low level language. This allows the Bytecode Verifier (BCV) of the Java platform to perform the static analysis of several safety policies. Because *proof-carrying code* is an emerging approach and a very promising technology (as witness recent initiatives like the European project MOBIUS IST 15905, the literature, or the emergence of certifying compilers (see the next section) for languages like JAVA), we will postpone its detailed description to the next section.

3 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage: www.igi-global.com/chapter/security-issues-mobile-code-paradigms/14077

Related Content

The Value of Coin Networks: The Case of Automotive Network Exchange
Andew Borchersand Mark Demski (2000). *Annals of Cases on Information Technology: Applications and Management in Organizations (pp. 109-123).*
www.irma-international.org/chapter/value-coin-networks/44631

Always-On Enterprise Information Systems with Service Oriented Architecture and Load Balancing
Serdal Bayram, Melih Kirlidogand Ozalp Vayvay (2010). *Information Resources Management: Concepts, Methodologies, Tools and Applications (pp. 850-867).*
www.irma-international.org/chapter/always-enterprise-information-systems-service/54520

The Effect of Task and Tool Experience on Maintenance CASE Tool Usage
Mark T. Dishawand Diane M. Strong (2003). *Information Resources Management Journal (pp. 1-16).*
www.irma-international.org/article/effect-task-tool-experience-maintenance/1245

IS Faculty Research Productivity: Influential Factors and Implications
Qing Huand T. Grandon Gill (2000). *Information Resources Management Journal (pp. 15-25).*
www.irma-international.org/article/faculty-research-productivity/1209

An Empirical Study of Technological Factors Affecting Cloud Enterprise Resource Planning Systems Adoption
Njenga Kinuthiaand Sock Chung (2017). *Information Resources Management Journal (pp. 1-22).*
www.irma-international.org/article/an-empirical-study-of-technological-factors-affecting-cloud-enterprise-resource-planning-systems-adoption/177189