CERIAS Tech Report 2001-46

On Mobile Code Security

Mohamed Hefeeda, Bharat Bhargava
Center for Education and Research in
Information Assurance and Security
&
Department of Computer Sciences, Purdue University
West Lafayette, IN 47907

On Mobile Code Security*

Mohamed Hefeeda and Bharat Bhargava
Center of Education and Research in Information Assurance and Security
And
Department of Computer Science, Purdue University
West Lafayette, IN 47907, U.S.A.
{mhefeeda, bb}@cs.purdue.edu

Abstract

While mobile agents approach provides a great flexibility and customizability compared to the traditional client-server approaches, it introduces many serious security problems. These problems are mainly protecting the hosting server and the visiting agent from each other. In this paper we discuss the security issues in the mobile agents technology. Specifically, we describe the techniques used to protect a hosting server from malicious agents roaming the network. We also present mechanisms for protecting a mobile agent during traveling from its source to the designated destination.

In addition, we address the challenging problem of securing the agent from a hostile execution environment. We point out the difficulty of the problem inherent from the fact that the executing environment has almost complete control over the code it is executing. And we describe the techniques proposed in the literature for dealing with the problem. Finally, we present a secure distributed application that we have developed to illustrate the capabilities of the mobile agents approach. We adapt a public key authentication technique to implement the security features of the application.

1 Introduction

Software agents are programs that act on behalf of their creators. *Mobile agents* are software agents that have the ability to travel from one place to another to do the work assigned to them autonomously. A mobile agent can interact with the hosts it visits and with the other agents that happen to be on those hosts at the same time.

Mobile agents paradigm has several advantages over other conventional network programming paradigms, such as Remote Procedure Call. Since agents carry objects, i.e. data and procedures, they can interact with the hosting servers without intensive communication between the servers and the clients that issued the agents. This will lower the network traffic required for the client/server applications. And hence, results in better network *performance* and *utilization*. Another advantage of mobile agents is the *automation* of a sequence of tasks. A user can dispatch an agent to carry out a sequence of tasks on different locations. These tasks may require the agent to travel to many places. Since the agent is autonomous, the user may need to be connected to the network only for a time long enough for the agent to leave his/her computer and perhaps sometime later when the agent returns from the trip carrying the results. A third advantage of mobile agents is their ability to distribute and update software packages.

* This research is sponsored in part by the National Science Foundation grants CCR-001712 and CCR-001788, CERIAS, and IBM SUR grant.

1.1 Mobile Agents Applications

Mobile agents have a potential for many distributed applications. Venners in [14] lists some of the applications that can benefit from the mobile agents technology. We summarize them as follows.

- ***Data collection from many places***. Mobile agents can be used to implement a network backup tool, for instance. The tool can employ an agent to periodically check and gather information from every computer in the network.
- ***Searching and filtering.*** A mobile agent could visit many sites, search through the information available at each site, and build an index of links to pieces of information that match a search criterion.
- ***Monitoring.*** An agent could go to a stock market host, wait for a certain stock to hit a certain price, and then notify its user or even buy some of the stocks on behalf of its user.
- ***Targeted information dissemination.*** Mobile agents can be used to distribute interactive news or advertisements.
- ***Negotiating.*** Agents could negotiate to establish a meeting time, get a reasonable price for a deal, and so on.
- ***E-Commerce.*** A mobile agent could do your shopping, including making orders and even paying.
- ***Parallel processing.*** Mobile agents could be used to distribute processes easily over many computers in the network.
- ***Entertainment.*** Mobile agents may represent game players. The agents compete with one another on behalf of the players. Each player would program an agent with a strategy, and then send the agent to a game host.

1.2 Mobile Agent Systems

A mobile agent system is an infrastructure that supports the mobile agent paradigm. It is the environment that the agents live in. There are currently several mobile agent systems. They differ in many aspects like: language used, security support, ease of developing applications, and so on. But they all provide the environment that the agents can live in and move from one site to another. Karnik in [4] gives a good comparison among some of the currently known mobile agent systems in terms of security features provided by each one of them.

1.3 Security Issues

Nothing is for free; as mobile agents paradigm facilitates network programming and distributed computing in general, it also introduces a lot of problems and challenges especially in the area of *security.* A mobile agent visiting a host will ask for some services and resources from the host. These resources should be protected from malicious or erroneous agents. Also, an agent carrying some private data such as credit card number or private key of its owner should protect these data from illegal access by the hosting server or other agents on that server. Not only the carried data needs to be protected but also sometimes we need to protect the code of the agent itself. Another important aspect of agent security is transferring the agent securely from its source to the desired destination. Simply because there may be some attackers listening to the network either to learn some of the information carried by the agent (passive attacks) or modifying that information for their favor (active attacks.)

The rest of this paper is organized as follows. Section 2 describes *three* different approaches (sandboxing, digital shrink-wrap, and proof-carrying code) used to protect the host from a malicious agent. Section 3 presents the means of protecting an agent from the active and passive attacks, which can happen while the agent is roaming around the network. The

challenging problem of protecting the mobile agent from a malicious host is described in section 4. Section 5 explains the details of the **Distributed Organizer** application. We conclude the paper in section 6.

2 Protecting the Host

The most obvious aspect with regard to the security of agent-based systems is how to protect the environment —in which the agent is supposed to execute— from hostile actions of a visiting mobile agent. There are various ways by which a malicious agent can harm the host. An agent may *steal* or manage to get illegal access to some private data, e.g. the financial data of a company from a database residing on the host. Another way of harming the host is by *damaging* or *consuming* the host resources; for example: delete some files, consume a lot of processing power, write enormous amount of data to the hard drive, or consume a lot of network bandwidth by establishing many connections with other servers.

Fortunately, this problem has taken a significant amount of research and it is almost solved. In the literature, there are basically *three* approaches that used to protect the host from a malicious code: *sandboxing [11]*, *digital shrink-wrap [8]*, and *proof-carrying code* [7]. We briefly describe these techniques in the following subsections.

2.1 Sandboxing

The idea of the sandbox model is that the host confines the visiting mobile code within a certain execution environment. In other words, the host allows the visiting code access to specific resources and prevents others from it. The sandbox can be customized to different sizes to fit the needs of different programs. Consider the following different-sized sandboxes [11], which illustrate the customizability of the approach.

- A sandbox that allows the program to access the CPU, the screen, keyboard, and mouse, and to its own memory. This is the minimal required sandbox for a program to run.
- A sandbox that allows the program to access the CPU, its own memory, and the web server from which it was loaded. This is usually called the default sandbox.
- A sandbox in which the program has access to the CPU, its own memory, its web server, and to a set of program-specific resources (local files, local machines, etc.).
- An open sandbox, in which the program is granted full access to whatever resources available from the host.

It is worth noting that the Java security model adopts the idea of sandboxing to *safely* execute Java applications (usually Applets) on a host (often within a web browser) [11].

2.2 Digital Shrink-Wrap

Another approach to protect an execution environment against potentially malicious mobile code is to *authenticate* the mobile code before it is actually executed [8]. In this approach, the producer of the code is required to *sign* it. And the code consumer verifies the signature of the producer before using it. Although it is not possible to decide whether a given piece of mobile code contains malicious code, one can at least determine whether it is authentically coming from its claimed source. Microsoft has proposed this approach in its Authenticode technology.

Note that the two approaches—sandboxing and digital shrink-wraps—can be combined to provide more sophisticated protection schemes. Actually, Sun Microsystems has combined both in its new Java 1.2 security model [11].

2.3 Proof-Carrying Code

Recently, Necula and Lee from Carnegie Mellon University propose a new technique to protect a host from a malicious mobile code [7]. The technique is called Proof-Carrying Code (PCC). PCC

enables a host to determine, automatically and with certainty that a program code provided by another system is safe to install and execute. The basic idea of PCC is that the *code producer* is required to provide an encoding of a proof that his/her code adheres to the security policy specified by the *code consumer*. The proof is encoded in a form that can be transmitted digitally. Therefore, the code consumer can quickly validate the code using a simple, automatic, and reliable proof-checking process.

Steps of generating and verifying the PCC

A typical PCC session requires *five* steps to generate and verify the PCC. The following figure shows an overview of the PCC process.

- ***Step 1***

  A PCC session starts with the code producer preparing the untrusted code to be sent to the code consumer. The producer adds *annotations* to the code, which can be done manually or automatically by a tool such as a certifying compiler. These annotations contain information that helps the code consumer to understand the safety-relevant properties of the code. The code producer then sends the annotated code to the code consumer to execute it.

- ***Step 2***

  The code consumer performs a fast but detailed inspection of the annotated code. This is accomplished using a program, called *VCGen*, which is one component of the consumer-defined safety policy. VCGen performs two tasks. First, it checks simple safety properties of the code. For example, it verifies that all immediate jumps are within the code-segment boundaries. Second, VCGen watches for instructions whose execution might violate the safety policy. When such an instruction is encountered, VCGen emits a predicate that expresses the conditions under which the execution of the instruction is safe. The collection of the verification conditions, together with some control flow information, make up the *safety predicate*, and a copy of it is sent to the proof producer.

- ***Step 3***

  Upon receiving the safety predicate, the proof producer attempts to prove it, and in the event of success it sends an encoding of a formal proof back to the code consumer. Because the code consumer does not have to trust the proof producer, any system can act as a proof producer.

- ***Step 4***

  Then, the code consumer performs a proof validation. This phase is performed using a *proof checker*. The proof checker verifies that each inference step in the proof is a valid instance of one of the axioms and inference rules specified as part of the safety policy. In addition, the proof checker verifies that the proof proves the same safety predicate generated in Step 2.

- ***Step 5***

  Finally, after the executable code has passed both the VCGen checks and the proof check, it is trusted not to violate the safety policy. It can thus be safely installed for execution, without any further need for run-time checking.

3 Protecting the Agent during the Transfer

As a mobile agent moves around the network, its code as well as its data is vulnerable to various types of security threats. There are two known types of attacks during transferring an agent from a server to another, namely: *passive attacks* and *active attacks.*

3.1 Passive Attacks

In passive attacks, an adversary attempts to extract some information from messages exchanged between two communicating parties without modifying the contents of the messages. Such attacks are often called eavesdropping. Usually *cryptographic mechanisms*, such as *RSA* and *ElGamal* cryptosystems [13, 6, 10] are used to protect against this kind of attacks.

3.2 Active Attacks

The active attack adversaries are more powerful than the passive attack adversaries. The adversary in this case is able to *modify* (tamper with) the data or the code of a mobile agent. The adversary is doing these modifications hopping that s/he might benefit from them. More dangerously, an adversary may impersonate a legitimate principal in the system and intercept messages intended for that principal. ***Data integrity*** mechanisms can be used to protect against tampering while the ***authentication*** mechanisms can be used to protect against impersonation.

Data integrity means that the data is either delivered intact or a flag is raised indicating that the data may be tampered with. This is usually accomplished using a *message digest* technique [6, 10, 13,]. The sender of the mobile agent appends to it a *digest* that can be generated *only* from the original contents of the mobile agent. Therefore, if an adversary tampered with the contents of the mobile agent, it would not be able to generate the correct digest for the new state of the mobile agent. Hence, the receiver of the mobile agent will easily detect the attack by checking the digest. The important point here is that, no two messages should have the same digest. Fortunately, the ***Collision-Free Hash Functions*** can be used to generate a unique digest for every message. A collision-free hash function takes a message of an arbitrary length and produces a unique digest of a specified fixed size. **MD5** message-digest algorithm, developed by Ron Rivest was the most widely used secure hash algorithm until the last few years. The advancement of computer technology has made security of **MD5** questionable because the brute-force attack can be used to break it. Therefore, the National Institute of Standards and Technology (NIST) has developed and published a more secure hashing algorithm, which is the Secure Hash Algorithm (**SHA**) [6, 10, 12].

4 Protecting the Agent

In general, it is very difficult to protect an executing program from the environment that is responsible for its execution. Therefore, protecting an agent is more difficult and challenging than protecting the host resources from a malicious agent [2, 3, 4, 8, 9, 15]. Let's first consider what a host can do for an agent. First, a host may simply destroy the agent and hence impede the function of its parent application. Second, It can steal sensitive information carried by the agent such as the results gathered so far or a private key of the agent's owner. Third, the host can modify the data carried by the agent for its favor. For instance, it might change the price quoted by another competitor. Moreover, a host may modify the agent's code to perform some dangerous actions when it returns to its home site.

In the literature, three approaches are proposed to protect a mobile agent from the host it is executing on: *limited blackbox security [2]*, *computing with encrypted functions [9]*, and *cryptographic traces [15]*. The rest of this section illustrates these approaches.

4.1 Limited Blackbox Security

Hohl from University of Stuttgart has introduced the idea of blackbox security to protect mobile code against malicious hosts [2]. The key idea of blackbox security is to generate an executable code from a given agent specification. This generated code is executed as a "blackbox" by the host, i.e. the host can not modify or read it but it only can execute it as is. Figure 1 describes the idea of the blackbox approach.

According to Hohl, currently there is no known algorithm to fully provide the blackbox protection as defined above. Therefore, the blackbox property needs to be relaxed a bit to enable practical implementation of the approach. The relaxation is mainly addressing the longevity of the blackbox property, i.e. how long the blackbox property should be valid. In the revised definition, it is not assumed that the blackbox protection holds forever, but *only* for a certain known time interval. According to this definition, an agent has the time-limited blackbox property if for a certain known time interval it cannot be attacked in the above-mentioned sense. To make the protection time interval explicit, an expiration time or date may be attached to the balckbox.

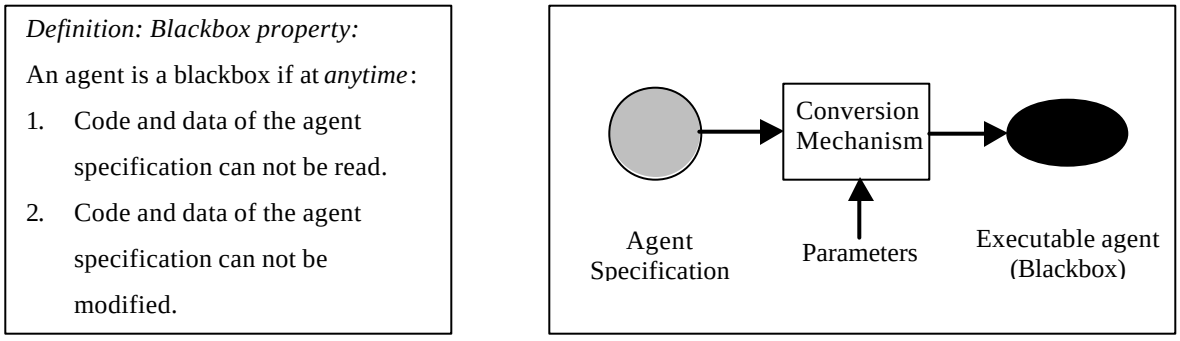

Figure 1. Blackbox Security

In order to achieve the limited blackbox property for mobile agents, Hohl has devised several conversion algorithms. In short, the task of a conversion algorithm is to generate a new agent out of an original agent, which differs in code and representation but yields the same results. In addition, the newly generated agent is assumed to be hard to analyze. In this context, "hard" means that the analysis required to understand the agent's functionality should take as much time as possible for an arbitrary attacker. Such conversion algorithms are sometimes also called *obfuscating* or *messing-up* algorithms. The reader should notice that the limited blackbox security does not assume that it is impossible for an attacker to analyze an agent, only that the analysis takes too much time.

Blackbox security does not protect against every possible attack. For example, it is still possible for the host to deny the execution or to return wrong system call results. Moreover, it is still possible for an attacker to read or to manipulate data and code, but as he cannot determine the role of these elements for the application, the attack results are random.

4.2 Computing with Encrypted Functions

Sander and Tschudin claim in [9] that the general belief about a mobile agent's vulnerabilities is wrong simply because it assumes that a mobile agent consists of plaintext code and data. They argue that there is no intrinsic reason why a program must be executed in a plaintext form. Therefore, one can have a computer executes a *cipher program* without understanding it. Similar to the problem of computing with encrypted data (CED) [1], they propose the *non-interactive computing with encrypted functions* (CEF) method as a general solution for the security requirements of mobile code. The problem is defined as follows [9]:

Alice has an algorithm to compute a function f. Bob has an input x and is willing to compute f(x) for her, but Alice wants Bob to learn nothing

substantial about f. Moreover, Bob should not need to interact with Alice during the computation of f(x).

They propose the following protocol as a general framework for computing with encrypted functions (CEF):

(1) Alice encrypts *f*.
(2) Alice creates a program *P(E(f))* which implements *E(f).*
(3) Alice sends *P(E(f))* to Bob.
(4) Bob executes *P(E(f))* at *x*.
(5) Bob sends *P(E(f))(x)* to Alice.
(6) Alice decrypts *P(E(f))(x)* and obtains *f(x).*

Obviously, the main obstacle here is how to efficiently encrypt a function. As they suggested, a polynomial function *f* can be encrypted by *composing* it with another polynomial *g*. Specifically, assume that *f* is a rational function (the quotient of two polynomials) and *s* is another rational function that Alice can invert efficiently. Then, the encryption of *f* is: $E(f) = s \circ f$. The security of the method is based on the difficulty of decomposing the resulting *E(f).* According to Zippel [16] there is no known polynomial time algorithm for decomposing multivariate rational functions.

Sander and Tschudin developed encryption schemes for polynomial and rational functions only but they could not yet provide such schemes for general functions.

4.3 Cryptographic Traces

Vigna in [15] has developed a mechanism to *detect* attacks against code, state, and control flow of mobile agents. His approach is different from the previous two approaches, which aimed at *preventing* attacks against an agent. The approach is based on execution tracing and cryptography that allows one to detect any possible illegal modification of agent's code, state, or execution flow.

The mechanism is based on post-mortem analysis of data (called traces) that are collected during the execution of an agent. The traces are then used as a basis for code execution verification, i.e. has the code executed its designated tasks properly or not? This way, in case of tampering, the agent's owner can prove that the claimed operations could have never been performed by the agent.

There are some limitations for this approach as pointed out by Vigna himself. First of all, the mechanism allows detection of an attack *after* the execution. Therefore, if a timely detection is needed, a different mechanism must be devised. Another limit stems from the fact that it is a *detection* mechanism, which means that it is useless unless there is a way to sue or punish the cheating sites or principals responsible for misbehaving agents. A third limit of the approach comes from the quantity of resources that are needed to enforce it. Because it makes extensive use of public-key cryptography algorithms, which are slow.

5 Distributed Organizer

In this section, we present a secure distributed application that we have developed to illustrate the capabilities of the mobile agents approach. This application also describes a way of employing cryptographic techniques to develop secure systems using mobile agents.

The **Distributed Organizer** is a distributed application used to schedule meetings among a group of users. Suppose that there is a group of users in which one of them needs to schedule a meeting with some or all of the other users. So, that user should communicate with the others to find a suitable common time for the meeting—a lengthy and time-consuming task. To save the time and effort of that user (might be the *boss*), the **Distributed Organizer** can automatically

accomplish this tedious task, i.e. finding a suitable time for the meeting and update the schedules of the users if it finds a match.

An interesting feature in the **Distributed Organizer** is its **security**. It allows only the *authorized* user to update the schedules of the other users. This is done by employing a **Public Key Authentication Protocol.** In this protocol, the user is asked to sign a randomly chosen message (nonce) using his own private key. Then, the host verifies the validity of the signature using the public key of the visiting user. The rest of this section describes the operation and implantation details of the system.

5.1 Operation of the Distributed Organizer

Basically, the Distributed Organizer is composed of two sets of agents: *Stationary Agents* and *Mobile Agents.* The *Stationary Agents* reside on the machines the participating while the *Mobile Agents* roam the network to collect and update the data from the stationary agents. The task of the application (i.e., scheduling a meeting time) is accomplished through *three* phases: collecting data, finding a common time, and updating the schedules.

Collecting data

If a user wants to schedule a meeting, s/he will ask her/his own stationary agent to perform that task. This stationary agent will dispatch a set of mobile agents, one for each participant in the meeting, to collect available time slots. Those agents will go (in parallel) and communicate with the remote stationary agents and get the schedules from them. After that, they will send these schedules back to their dispatcher. This step is illustrated in figure 2.

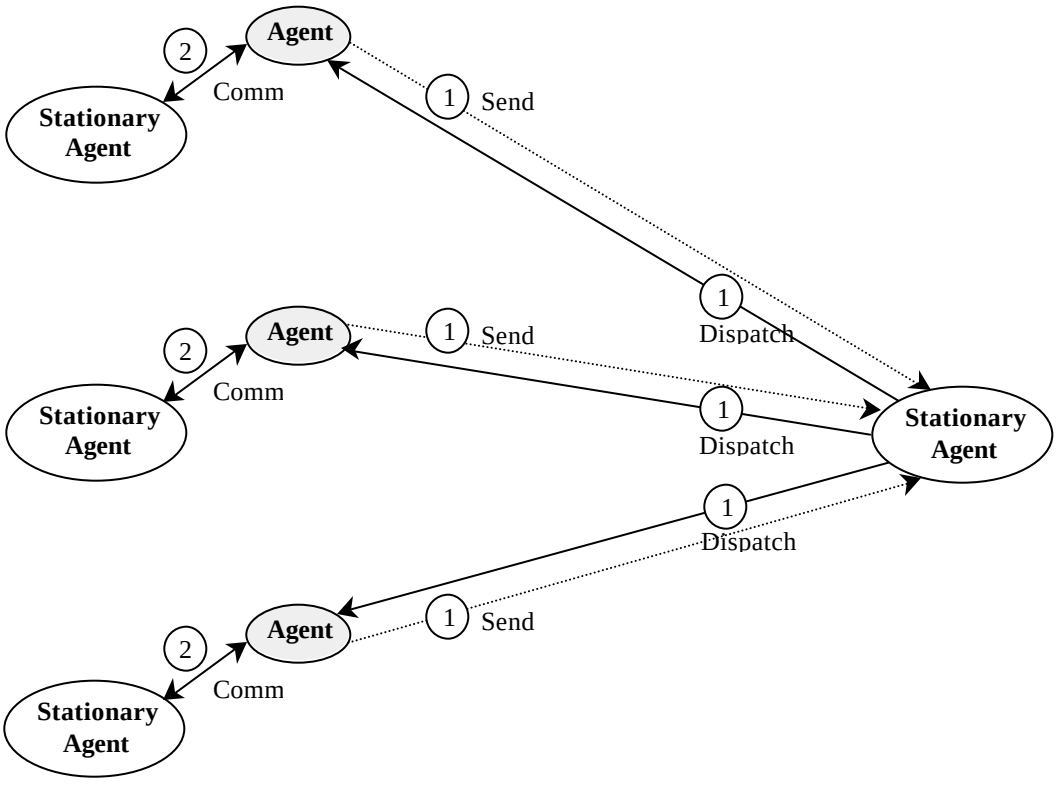

Figure 2. The Stationary Agent dispatches mobile agent s to collect data

Finding a common time

After the stationary agent gets all the required schedules from the remote agents, it tries to find a common suitable time for the meeting. If it doesn't find any suitable time that matches the options that the user specified, it terminates with a message *"Couldn't schedule a meeting at the given time options."* Otherwise, it proceeds to phase 3, updating the schedules.

Updating the schedules

Now, we found a common time slot for the meeting. So we need to mark or reserve this time slot in the schedule of all participants. Thus, the stationary agent dispatches another group of mobile agents (*updating agents*) to update the schedules of the remote users. The remote stationary agents require that the updating agents *authenticate* themselves before updating the schedules. A ***public key authentication protocol*** based on the ***Digital Signature Algorithm (DSA)*** [12] is used during this step. Specifically, the remote stationary agent generates a random message (step 1 in the figure) and sends it to the updating agent to sign it (step 2). The updating agent sends this message to its parent who signs it using the private key and sends it back to the updating agent (steps 3-6). Then, the updating agent sends the signature to the stationary agents who verifies it using the public key of the updating agent's owner (steps 7-9). The updating process and the *authentication* process between the updating agent and the remote stationary agent are depicted figure 3.

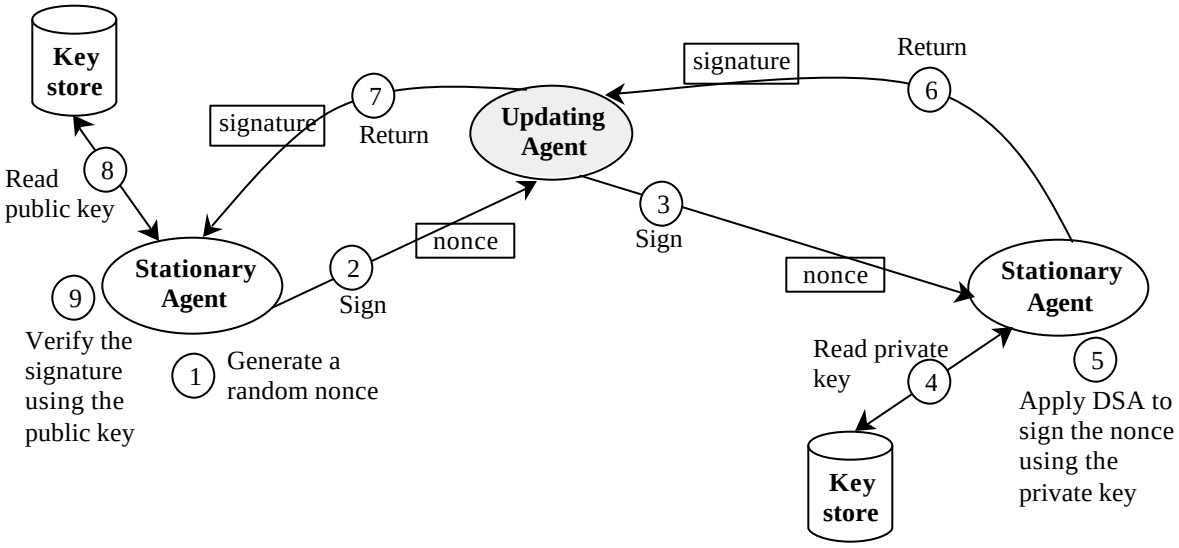

Figure 3. The Authentication Process in the Distributed Organizer

Comments:

- The random message (nonce) is generated using a *cryptographic secure pseudo-random generator*, which is provided by the Java language (`java.security.secureRandom` class.)
- The Key Store is a database that stores the private keys as well as the certificates of the users. The certificate contains some information about a certain user along with his/her public key. Java 1.2 comes with a tool (`keytool`) to create and manage the Key Store. Moreover, Java

1.2 has a class (java.`security`.KeyStore) which facilitates the communication between a Java program and the `KeyStore`. The `KeyStore` database allows a user to access the certificates (and hence, the public keys) of other users. But it requires a password to access the private keys. So, it is a secure method to distribute the keys within a small to moderate size computing environments. We've implemented and tested the authentication protocol based on the KeyStore. Unfortunately, Aglets Work Bench doesn't support Java 1.2. So, we couldn't use the idea of the KeyStore. Therefore, we had to use another way to distribute the keys. Basically, we implemented a simple `KeyStore`. We implemented a class (`GenerateKeys`) which generates a private key and public key (not a certificate) for each user. Then, it stores (using Java Object Serialization) the private key in a file and the public key in another file. Each user's private key file is stored in his home (private) directory, while all public key files are stored in a shared (read only) directory. Each public key file is named by its user name; for example, a file named *bob* contains the public key of *bob*.

5.2 Implementation

We used Java language and IBM Aglets Work Bench [5, 18] to implement the project. The project consists of the following classes:

1. GenerateKeys

This class is used to generate pair of Keys and save them in two files. One file contains the private keys and will be saved locally and safely by each user. The other file contains the public key of the user. All public key files will be distributed to all uses beforehand or they will be stored in a shared (read only) directory.

2. Server

This is a stationary agent which resides on the machine of the party who is scheduling the meeting (say the Boss). First, It creates mobile agents and dispatches them to collect the schedules of the involved parties. After it gets all schedules, it tries to find a common available time. Then, if it finds a suitable time for all of them, it dispatches updating agents (CalanderUpdate) to mark the schedules of all parties after "Authenticating themselves".

3. CalanderServer

This is another Stationary Agent that resides on the machine of each participating party. It reads the schedule from a file stored locally. Then, it interacts with the visiting mobile agents. First, it interacts with the agents that request the schedule (without authentication, actually no need for authentication here since it takes long time). Then, it interacts with the updating agent that needs to mark a certain time for the meeting. But before updating, the stationary agent asks the updating agent (CalanderUpdate) to authenticate itself. This is done by generating a random message (nonce) and asking the updating agent to ***sign*** it.
Then, the stationary agent verifies the signature using the public key of the owner of the updating agent.

4. CalanderAgent

This is a mobile Agent that visits the involved parties and gets their schedules. Then, it sends the schedules to its parent (creator). This agent communicates with the stationary agents on the host.

5. CalanderUpdate

This is another mobile agent used to update the schedule after finding a common meeting time. This agent communicates with the stationary agent on the host first to verify itself then to update the schedule. First, It authenticates itself by sending the randomly generated message (which

generated by the hosting stationary agent) to its parent to sign it. When it gets the signature from the parent, it passes it to the stationary agent who checks it using the public key of the owner of the updating agent.

6 Conclusions

In this paper, we shed the light on the security issues in the mobile agents technology. We described the techniques used to protect a hosting server from potential attacks by malicious agents roaming the network. We also presented the cryptographic mechanisms (encryption and secure hashing) used to protect a mobile agent during its travel from active and passive attacks.

In addition, we addressed the challenging problem of securing the agent from a hostile execution environment. We pointed out the difficulty of the problem inherent from the fact that the executing environment has almost complete control over the code it is executing. We described the techniques proposed so far in the literature for dealing with the problem. Specifically, we described the blackbox security, computing with encrypted functions, and cryptographic traces. All of these techniques *partially* solve the difficult problem and none of them is general enough to handle the entire problem. The most promising approach, from our point of view, is the Computing with Encrypted Functions (CEF) approach.

Finally, we presented a secure distributed application that we have developed to illustrate the capabilities of the mobile agents approach. We adapted a public key authentication technique to implement the security features of the application. The application, which we call the **Distributed Organizer,** is used to organize the agenda of a group of users. Mainly, it is used to conveniently and quickly schedule meetings among the participating parties.

7 References

[1] M. Abadi and J. Feigenbaum, "Secure Circuit Evaluation," Journal of Cryptology, 2(1), 1990, pp.1-12.

[2] F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts," in: G.Vigna, editor, *Mobile Agents and Security, Lecture Notes in Computer Science 1419*, Springer-Verlag, Berlin, 1998, pp. 92-113.

[3] G. Karjoth, D. B. Lange, and M. Oshima, "A security Model for Aglets," *IEEE Internet Computing*, July-August 1997, pp.68-77.

[4] N. Karnik, *Security in Mobile Agent Systems*, Ph.D. dissertation, Department of Computer Science and Engineering, University of Minnesota, 1998.

[5] D. B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesely, 1998.

[6] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Inc., Boca Raton, FL, 1996.

[7] G.C. Necula and P. Lee, "Safe, Untrusted Agents Using Proof-Carrying Code," in: G.Vigna, editor, *Mobile Agents and Security, Lecture Notes in Computer Science 1419*, Springer-Verlag, Berlin, 1998, pp. 61-91.

[8] R. Oppliger, "Security issues related to mobile code and agent-based systems," *Computer Communications* 22, 1999, pp. 1165-1170.

[9] T. Sander and C. Tschudin, "Protecting Mobile Agents Against Malicious Hosts," in: G.Vigna, editor, *Mobile Agents and Security, Lecture Notes in Computer Science 1419*, Springer-Verlag, Berlin, 1998, pp. 44-60.

[10] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, New York, 2nd edition, 1996.

[11] Scott Oaks, *Java Security*, O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

[12] W. Stallings, *Cryptography and Network Security: Principles and Practice*, Prentice-Hall, Inc., New Jersey, 2nd edition, 1999.
[13] D. Stinson, *Cryptography: Theory and Practice*, CRC press, Inc., Boca Raton, FL, 1995
[14] B. Venners, "Solve Real Problems with Aglets, a Type of Mobile Agent," *JavaWorld*, http://www.javaworld.com/javaworls/jw-05-1997/jw-05-hood.html.
[15] G. Vigna, "Cryptographic Traces for Mobile Agents," in: G.Vigna, editor, *Mobile Agents and Security*, *Lecture Notes in Computer Science 1419*, Springer-Verlag, Berlin, 1998, pp. 137-153.
[16] R. E. Zippel, "Rational Function Decomposition," *In proceedings of the International Symposium on Symbolic and Algebraic Computation*, ACM Press, July 1991, pp. 1-6.
[17] http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html.
[18] http://www.trl.ibm.co.jp/aglets