# AN OVERVIEW OF SECURITY ISSUES AND TECHNIQUES IN MOBILE AGENTS

Mousa Alfalayleh and Ljiljana Brankovic
*The School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle, NSW 2308, Australia. E-mail: {mousa, lbrankov}@cs.newcastle.edu.au*

**Abstract**: Mobile agents are programs that travel autonomously through a computer network in order to perform some computation or gather information on behalf of a human user or an application. In the last several years, mobile agents have found numerous applications including e-commerce. In most applications, the security of mobile agents is of the utmost importance. This paper gives an overview of the main security issues related to the mobile agent paradigm. These issues include security threats, requirements, and techniques for keeping the mobile agent platform and the agent itself secure against each other.

**Key words**: Security, Mobile agents, Mobile code, Malicious host, Electronic commerce.

## 1. INTRODUCTION

During the last several years, we have witnessed fundamental changes in distributed and client-server computer systems. In the past, software applications were bound to particular nodes in computer networks. This reality has changed with the appearance of mobile agents [1], that is, programs that act in a computer network on behalf of a human user or an application. Agents can travel autonomously among different nodes in the network, in order to perform some computation or gather information [2]. In this paradigm, clients do not need to have a network connection established while their agents are performing operations on different servers. As such, they provide an appealing alternative to the client-server architecture for many applications [3].

The applications of mobile agent technology are abundant and include electronic commerce, personal assistance, distributed information search and

retrieval, monitoring, network management [4], real-time control, building middleware services, military command and control [5], and parallel processing. The promises made by this technology can hardly be overstated. There are numerous advantages of using the mobile agent paradigm rather than conventional paradigms such as client-server based technologies. Using a mobile agent paradigm reduces network usage [2], dynamically updates server interfaces, improves fault tolerance [6], introduces concurrency [2], and assists operating in heterogeneous environments [4].

On the other hand, mobile agent technology has some limitations, primarily in the area of security. These limitations have raised many concerns about the practical utilisation of mobile agents. Current research efforts in the area of mobile agent security adopt two different points of view. Firstly, from the platform perspective, we need to protect the host from malicious mobile agents such as viruses and Trojan horses that are visiting it and consuming its resources. Secondly, from the mobile agent point of view, we need to protect the agent from malicious hosts. Both points of view have attracted much research effort. This paper gives an overview of the main solutions that have been described in the literature to keep the mobile agent platform and the agent itself protected from each other.

The paper is organized as follows. Section 2 deals with the security issues related to the mobile agent paradigm such as security threats and requirements. Section 3 gives an overview of the main solutions for keeping a mobile agent platform secure against a malicious mobile agent. Similarly, Section 4 presents a set of solutions for ensuring the security of mobile agents against illegitimate platforms. Finally, Section 5 gives some concluding remarks.

## 2. SECURITY ISSUES IN THE MOBILE AGENT PARADIGM

The mobile agent paradigm appeals to many specialists working in different applications. This is especially true for e-commerce applications, including stock markets and electronic auctions. Such applications involve dealing with vast amounts of money and thus users will hesitate to use mobile agents unless they feel that they are secure and can be trusted. Therefore, the security of mobile agents is an important issue that has triggered much research effort in order to find a suitable solution.

One of the most valuable characteristics of mobile agents is their mobility that enables them to travel autonomously through the network. However, it is precisely because of this property that mobile agents are

exposed to different types of attacks. We next present these attacks, together with those that are launched by agents to harm platforms.

**Unauthorized Access.** Malicious mobile agents can try to access the services and resources of the platform without adequate permissions. In order to thwart this attack, a mobile agent platform must have a security policy specifying the access rules applicable to various agents, and a mechanism to enforce the policy.

**Masquerading.** In this attack, a malicious agent assumes the identity of another agent in order to gain access to platform resources and services, or simply to cause mischief or even serious damage to the platform. Likewise, a platform can claim the identity of another platform in order to gain access to the mobile agent data. In both cases, the malicious agent or platform will not receive any blame for its potentially detrimental actions. Instead, the unsuspecting agent or platform whose identity was misused will be held responsible [2,4].

**Denial of Service.** A malicious platform can cause harm to a visiting mobile agent by ignoring the agent's request for services and resources that are available on the platform, by terminating the agent without notification, or by assigning continuous tasks to the agent so that it will never reach its goal. Likewise, a malicious agent may attempt to consume the resources of the platform, such as disk space or processing time, or delete important files or even the whole hard disk contents, thus causing harm to the platform and launching a denial of service attack against other visiting agents [2,4].

**Annoyance attack.** Examples of this attack include opening many windows on the platform computer or making the computer beep repeatedly [2]. Such attacks may not represent a very serious problem to the platform, however they still need to be prevented.

**Eavesdropping.** In this attack, a malicious platform monitors the behavior of a mobile agent in order to extract sensitive information from it. This is typically used when the mobile agent code and data are encrypted. Monitoring may include the identity of the entities that mobile agent is communicating with, and the types of services requested by the mobile agent [2,4].

**Alteration.** In the alteration attack, a malicious platform tries to modify mobile agent information, by performing an insertion, deletion and/or alteration to the agent's code, data, and execution state. Modifying the mobile agent execution code and state may result in the agent performing harmful actions to other platforms, including the agent's home platform [2,4].

We next explore the different security requirements that the mobile agent paradigm needs to satisfy.

**Confidentiality.** It is important to ensure that the information carried by a mobile agent or stored on a platform is accessible only to authorized parties. This is also the case for the communication among mobile agent paradigm components.

**Integrity.** It is essential to protect the mobile agent's code, state, and data from being modified by unauthorized parties. This can be achieved either by preventing or by detecting unauthorized modifications.

**Availability.** Platforms typically face a huge demand for services and data. In the case that a platform cannot meet mobile agents' demands, it should notify them in advance. Additionally, a platform must be able to afford a certain level of fault-tolerance and fault-recovery from unpredicted software and hardware failures [4].

**Accountability.** Platforms need to establish audit logs to keep track of all visiting mobile agents' actions in order to keep them accountable for their actions. Audit logs are also necessary when the platform needs to recuperate from a security penetration or a system failure.

**Anonymity.** As mentioned above, platforms need to keep track of mobile agents' actions for accountability purposes. However, platforms also have to balance between their needs for audit logs and mobile agents' needs to keep their actions private [4].

In the next two sections we present the existing techniques for protecting agents and platforms. These techniques fall into two categories: Prevention and detection. Prevention techniques are aimed at making it impossible for platforms and agents to successfully perform an attack. For example, a tamper-proof device can be used to execute an agent in a physically sealed environment. However, in the literature the term "prevention mechanism" is often used to denote a technique that makes it impossible to modify an agent in a meaningful way [26]. Examples of such techniques include "Environmental Key Generation" and "Computing with Encrypted Functions". On the other hand, detection techniques aim at detecting the attacks. The "Co-Operating Agents" technique and "Execution Tracing" belong to this category.

## 3. SECURITY OF PLATFORMS

The primary issue in the security of mobile agent systems is to protect mobile agent platforms against malicious attacks launched by the agents. This section presents a set of detection and prevention techniques for keeping the platform secure against a malicious mobile agent.

## 3.1 Sandboxing

*Sandboxing* [7] is a software technique used to protect mobile agent platform from malicious mobile agents. In an execution environment (platform), local code is executed with full permission and has access to crucial system resources. On the other hand, remote code, such as mobile agents and downloadable applets, is executed inside a restricted area called a "sandbox" [10,11]. Restriction affects certain code operations [9] such as interacting with the local file system, opening a network connection, accessing system properties on the local system, and invoking programs on the local system. This ensures that a malicious mobile agent cannot cause any harm to the execution environment that is running it. A Sandboxing mechanism enforces a fixed security policy for the execution of the remote code. The policy specifies the rules and restrictions that mobile agent code should confirm to. A mechanism is said to be secure if it properly implements a policy that is free of flaws and inconsistencies [8].

The most common implementation of Sandboxing is in the Java interpreter inside Java-enabled web browsers. A Java interpreter contains three main security components: ClassLoader, Verifier, and Security Manager [8,11,12,13,16]. The ClassLoader converts remote code into data structures that can be added to the local class hierarchy. Thus every remote class has a subtype of the ClassLoader class associated with it [8]. Before the remote code is loaded, the Verifier performs a set of security checks on it in order to guarantee that only legitimate Java code is executed [12,13]. The remote code should be a valid virtual machine code, and it should not overflow or underflow the stack, or use registers improperly [8,16]. Additionally, remote classes cannot overwrite local names and their operations are checked by the Security Manager before the execution. For example, in JDK 1.0.x, classes are labelled as local and remote classes. Local classes perform their operations without any restrictions while remote classes should first surrender to a checking process that implements the platform security policy. This is implemented within the Security Manager. If a remote class passes the verification, then it will be granted certain privileges to access system resources and continue executing its code. Otherwise, a security exception will be raised [8,11,12,13,16].

A problem with the Sandboxing technique is that a failure in any of the three above mentioned interrelated security parts may lead to a security violation. Suppose that a remote class is wrongly classified as a local class. Then this class will enjoy all the privileges of a local class. Consequently, the security policy may be violated [8]. A downside of the Sandboxing technique is that it increases the execution time of legitimate remote code [7]

but this can be overcome by combining Code Signing and Sandboxing, as will be explained later.

## 3.2 Code Signing

The "Code Signing" technique ensures the integrity of the code downloaded from the Internet. It enables the platform to verify that the code has not been modified since it was signed by its creator. Code Signing cannot reveal what the code can do or guarantee that the code is in fact safe to run [14,15].

Code Signing makes use of a digital signature and one-way hash function. A well-known implementation of code signing is Microsoft Authenticode, which is typically used for signing code such as ActiveX controls and Java applets [15].

Code Signing enables the verification of the code producer's identity but it does not guarantee that they are trustworthy. The platform that runs mobile code maintains a list of trusted entities and checks the code against the list. If the code producer is on the list, it is assumed that they are trustworthy and that the code is safe. The code is then treated as local code and is given full privileges; otherwise the code will not run at all. This is known as a "black-and-white" policy [8,16], as it only allows the platform to label programs as completely trusted or completely untrusted.

There are two main drawbacks of the Code Signing approach. Firstly, this technique assumes that all the entities on the trusted list are trustworthy and that they are incorruptible. Mobile code from such a producer is granted full privileges. If the mobile agent is malicious, it can use those privileges not only to directly cause harm to the executing platform but also to open a door for other malicious agents by changing the acceptance policy on the platform. Moreover, the affects of the malicious agent attack may only occur later, which makes it impossible to establish a connection between the attack and the attacker [8]. Such attacks are referred to as "delayed attacks". Secondly, this technique is overly restrictive towards agents that are coming from untrustworthy entities, as they do not run at all. The approach that combines Code Signing and Sandboxing described in the next section alleviates this drawback.

## 3.3 Code Signing and Sandboxing Combined

Java JDK 1.1 combines the advantages of both Code Signing and Sandboxing. If the code consumer trusts the signer of the code, then the code will run as if it were local code, that is, with full privileges being granted to it. On the other hand, if the code consumer does not trust the signer of the

code then the code will run inside a Sandbox as in JDK1.0 [17,21]. The main advantage of this approach is that it enables the execution of the mobile code produced by untrustworthy entities. However, this method still suffers from the same drawback as Code Signing, that is, malicious code that is deemed trustworthy can cause damage and even change the acceptance policy.

The security policy is the set of rules for granting programs permission to access various platform resources. The "black-and-white" policy only allows the platform to label programs as completely trusted or untrusted, as is the case in JDK1.1. The combination of Code Signing and Sandboxing implemented in JDK 1.2 (Java 2) incorporates fine-grained access control and follows a "shades-of-grey" policy. This policy is more flexible than the "black-and-white" policy, as it allows a user to assign any degree of partial trust to a code, rather than just "trusted" and "untrusted" [16,17]. There is a whole spectrum of privileges that can be granted to the code. In JDK1.2 all code is subjected to the same security policy, regardless of being labelled as local or remote. The run-time system partitions code into individual groups called protection domains in such a way that all programs inside the same domain are granted the same set of permissions. The end-user can authorize certain protection domains to access the majority of resources that are available at the executing host while other protection domains may be restricted to the Sandbox environment. In between these two, there are different subsets of privileges that can be granted to different protection domains, based on whether they are local or remote, authorised or not, and even based on the key that is used for the signature [16,17,18]. Although this scheme is much more flexible than the one in JDK 1.1, it still suffers from the same problem, that an end user can grant full privileges to malicious mobile code, jeopardising the security of the executing platform.

## 3.4 Proof-Carrying Code

Lee and Necula [19] introduced the *Proof-Carrying Code* (PCC) technique in which the code producer is required to provide a formal proof that the code complies with the security policy of the code consumer. The code producer sends the code together with the formal safety proof, sometimes called machine-checkable proof, to the code consumer. Upon receipt, the code consumer checks and verifies the safety proof of the incoming code by using a simple and fast proof checker. Depending on the result of the proof validation process, the code is proclaimed safe and consequently executed without any further checking, or it is rejected [4,19,21,22]. PCC guarantees the safety of the incoming code providing that there is no flaw in the verification-condition generator, the logical axioms, the typing rules, and the proof-checker [20].

PCC is considered to be "self-certifying", because no cryptography or trusted third party is required. It involves low-cost static program checking after which the program can be executed without any expensive run-time checking. In addition, PCC is considered "tamper-proof" as any modification done to the code or the proof will be detected. These advantages make the Proof Carrying Code technique useful not only for mobile agents but also for other applications such as active networks and extensible operating systems [19,22].

Proof Carrying Code also has some limitations, which need to be dealt with before it can become widely used. The main problem with PCC is the proof generation, and there is a lot of research on how to automate the proof generation process. For example, a certifying compiler can automatically generate the proof through the process of compilation [19,23]. Unfortunately, at present many proofs still have to be done by hand [21]. Other limitations of the PCC technique include the potential size of the proof and the time consumed in the proof-validation process [19].

## 3.5 State Appraisal

While a mobile agent is roaming among agent platforms, it typically carries the following information: code, static data, collected data, and execution state. The execution state is dynamic data created during the execution of the agent at each platform and used as input to the computations performed on the next platform. The state includes a program counter, registers, local environment, control stack, and store. The state of a mobile agent changes during its execution on a platform. Farmer et al [25] introduced the "State Appraisal" technique to ensure that an agent has not become malicious or modified as a result of its state alterations at an untrustworthy platform.

In this technique the author, who creates the mobile agent, produces a state appraisal function. This function calculates the maximum set of safe permissions that the agent could request from the host platform, depending on the agent's current state. In other words, the author needs to anticipate possible harmful modifications to the agent's state and to counteract them within the appraisal function. Similarly, the sender, who sends the agent to act on his behalf, produces another state appraisal function that determines the set of permissions to be requested by the agent, depending on its current state and on the task to be completed. Subsequently, the sender packages the code with these state appraisal functions. If both the author and the sender sign the agent, their appraisal functions will be protected against malicious modifications. Upon receipt, the target platform checks and verifies the correct state of the incoming agent. Depending on the result of the

verification process, the platform can determine what privileges should be granted to this incoming agent given its current state. Clearly, when the author and the sender fail to anticipate certain attacks, they cannot include them in the appraisal functions and provide the necessary protection [4,24,25].

In addition to ensuring that an agent has not become malicious during its itinerary, the State Appraisal may also be used to disarm a maliciously altered agent [25]. Another advantage of this technique is that it provides a flexible way for an agent to request permissions depending on its current state and on the task that it needs to do on that particular platform [24,25]. The main problem with this technique is that it is not easy to formulate appropriate security properties for the mobile agent and to obtain a state appraisal function that guarantees those properties [24].

## 3.6 Path Histories

When an agent travels through a multi-hop itinerary, it visits many platforms that are not all trusted to the same extent. The newly visited platform may benefit from the answers to the following questions: Where has the agent been? How likely is it that the agent has been converted to a malicious one during its trip? To enable the platform to answer these questions, a mobile agent should maintain an authenticable record of the previously visited platforms during its travel life. Using this history, the platform makes the decision whether to run the agent and what level of trust, services, resources and privileges should be granted to the agent [4,26,27]. The list of the platforms visited previously by the agent is the basis of trust that the execution platform has in the agent. Typically, it is harder to maintain trust in agents that have previously visited a huge number of platforms. Likewise, it is harder to trust the agent whose travel path is unknown in advance, for example the agent that is searching for new information and creates its travel path dynamically [27].

The "Path History" is constructed in the following way. Each visited platform in the mobile agent's travel life adds a signed record to the Path History. This record should contain the current platform's identity together with the identity of the next platform to be visited in the mobile agent's travel path. Moreover, in order to prevent tampering, each platform should include the previous record in the message digest that it is signing [4]. After executing the agent, the current platform should send the agent together with the complete Path History to the next platform. Depending on the information in the Path History, the new platform can decide whether to run the agent and what privileges should be granted to the agent. The main problem with the Path History technique is that the cost of the path

verification process increases with the path history [4,26,27]. Constructing algorithms for Path History evaluation is an interesting research area [27].

# 4. SECURITY OF MOBILE AGENTS

In the previous section, we presented several techniques for protecting mobile agent platforms against malicious mobile agents. On the other hand, mobile agents themselves are exposed to various threats by the platforms they visit.

## 4.1 Co-Operating Agents

In order to improve the security of mobile agents against the attacks that are launched by the malicious platforms, the Co-Operating Agent technique [28,29,4] distributes critical tasks of a single mobile agent between two co-operating agents. Each of the two co-operating agents executes the tasks in one of two disjoint sets of platforms. The co-operating agents share the same data and exchange information in a secret way. The Co-Operating Agent technique reduces the possibility of the shared data being pilfered by a single host. Each agent records and verifies the route of its co-operating agent [28,29]. Co-Operating Agents can be used to perform e-commerce tasks or protocols such as the authorization of negotiation, bidding, auction, electronic payment, etc [29,30].

When the agent travels from one platform to another, it uses an authenticated communication channel to pass information to its co-operating agent. The information includes details about the agent's itinerary such as the last platform visited by the agent, the current platform, and the next platform to be visited. The peer agent takes a suitable action when anything wrong occurs, e.g., a platform sends the agent to a wrong destination, or claims to have received the agent from an incorrect source. However, this technique has some drawbacks. One of them is the cost of setting up the authenticated communication channel for each migration. Another drawback is that in the case of a co-operating agent being killed, it is difficult for its peer to decide which platform is responsible [4,28,29].

It is worth noting that an assumption made in the Co-Operating Agent technique, is that only a small percentage of platforms are in fact malicious and that it is not very likely that both agents will encounter such a host. However, care should be taken that the two sets of platforms assigned to the two agents are indeed disjoint, that is, that they never encounter the same host. This method can easily be extended to more than two co-operating agents.

## 4.2 Execution Tracing

Execution Tracing enables detection of any possible misbehavior by a platform, that is, improper modification of the mobile agent code, state, and execution flow. This technique is based on cryptographic traces that are collected during an agent's execution at different platforms. Traces are logs of the actions performed by a mobile agent during its lifetime. Execution Tracing enables an agent's owner to check the agent's execution history and see if it contains any unauthorized modifications done by a malicious platform. Each trace contains identifiers of all the statements performed on a particular platform. In the case that some of the statements require information from the external execution environment, the trace must also contain a digital signature of the platform. Such statements are known as "black" statements. On the other hand, the statements that only use the values of the agent's internal variables are called "white" statements [31,32].

The Execution Tracing technique assumes that all the involved parties own a public and private key that can be used for digital signatures, in order to identify involved parties. Different parties, such as users and platform owners, communicate by using signed messages. A platform that receives the agent and agrees to execute it produces the associated trace during the agent's execution. The message that an execution platform attaches to the mobile agent typically contains information such as the unique identifier of the message, the identity of the sender, the timestamp, the fingerprint of the trace, the final state and the trusted third party (which could later be used to resolve disputes). Later, the owner of the agent may suspect that a certain platform cheated while executing the agent. If this is the case, the owner will ask the suspicious platform to reproduce the trace. Finally, the agent's owner validates the execution of the agent by comparing the fingerprint of the reproduced trace against the fingerprint of the trace that is originally supplied by the suspicious platform [31].

In addition to detection of any modification of the agent performed by a malicious platform, Execution Tracing also provides a means to protect a legitimate platform against a malicious agent by obtaining the related traces from the involved parties. Execution Tracing has some limitations, such as the potential large size and number of logs to be retained. Another limitation of this technique is that the owner platform needs to wait until it obtains suspicious results in order to run the verification process. Also, this technique is considered to be too difficult to use in the case of multi-threaded agents [31,32].

A new version of the Execution Tracing technique, proposed by Tan and Moreau [32,33], modifies the original technique by assigning the trace

verification process to a trusted third party, the verification server, instead of depending on the agent's owner.

When a mobile agent travels to a new platform during its itinerary, a copy of the agent is submitted to a corresponding verification server. The visited platform receives the agent and produces the associated execution trace. Before the agent's migration from the current platform to a new one, the current platform forwards the trace to a corresponding verification server. The verification server simulates the execution of the agent on the platform by using the corresponding execution trace and the agent's copy. The simulation process is repeated for every platform in the agent's path by the corresponding verification server, until the agent is sent back to its home platform. Tan and Moreau [32] provided a detailed protocol of message exchanges, as well as the formal modeling and verification of the protocol.

Execution Tracing with a verification server does not wait until a suspicion is raised in order to run the verification process. The verification here is compulsory and this is an advantage over the original Execution Tracing technique where the verification process is triggered only by suspicious results [32]. However, Execution Tracing with a verification server still suffers from the same limitation as the original technique, that is, the need to retain a potentially large size and number of logs. Additionally, each platform chooses a verification server and that might encourage and facilitate a possible malicious collaboration between a platform and the server.

## 4.3 Environmental Key Generation

Riordan and Schneier [34] designed the Environmental Key Generation technique to be used when a platform wants to communicate with another platform by sending it a message, yet it only wants the receiving platform to obtain the message if some environmental condition is satisfied. This can be achieved by sending a mobile agent carrying an encrypted message. The encrypted message may include some data and/or executable code. Neither can the mobile agent precisely predict its own execution at the receiver platform, nor can the platform foresee the incoming agent task. The agent will wait at the receiving platform for some environmental condition to occur. The environmental condition could be, for example, matching a certain search string. When the environmental condition is met, an activation key is generated in order to decrypt the enciphered message that the mobile agent is carrying. Without meeting the environmental condition, the agent is unable to decrypt its own message [34].

The activation key, which is used to decrypt the agent's message, could be hidden inside a fixed data channel. If this data channel is, for example,

a file system, then the activation key could be hidden in a file or could be the hash of a certain file name. On the other hand, if the data channel is a mail message, the activation key could be a string inside this message or a hash of the message [34].

Environmental Key Generation may suit some applications other than mobile agents (some of which may even be malicious) including blind search engines, logic bombs, directed viruses, and remote alarms [34]. Tschudin [35] exploited the idea of Environmental Key Generation for the purpose of the programmed death of a mobile service, that is, the self-destruction of a mobile service when it is no longer required [35]. However, this technique has some limitations. The receiving platform could act maliciously against the incoming agent. When the environmental condition is met and the activation key is generated, the platform could modify the agent to perform a different function, for example, to print out the executable code instead of running it [4]. Another limitation of the technique is that the platform may consider it unsafe to execute an encrypted code that is attached to a mobile agent, as it could be, for example, a virus.

## 4.4 Non-Interactive Computing with Encrypted Functions

This technique represents a software solution for protecting a mobile agent from a malicious executing platform during its itinerary. This is a cryptographic solution to achieve integrity and privacy of the mobile agent. Protecting integrity means that the mobile agent is made safe against tampering by a malicious platform. Achieving privacy means that the mobile agent can conceal its program (code) when it is executed remotely in an untrusted environment. In addition to this, a mobile agent can safely compute cryptographic primitives on a remote platform by using this approach. An example of cryptographic primitives is a digital signature or encryption.

This technique is based on executing a program embodying an encrypted function on a mobile agent platform. It also ensures that the platform does not learn anything substantial about the encrypted function. Abadi and Feigenbaum [38] suggested the initial version of this technique. Their solution was interactive and required several rounds of message exchange with the agent's home platform. However, the interactive solution does not suit the mobile agent scenario, as agents operate autonomously without much interaction with their home platform.

Sander and Tschudin [36,37] suggested a non-interactive solution, which is suitable for the mobile agent paradigm. In their solution, the home platform has an algorithm to compute a function $f$. The target platform has

an input $x$ and can provide a service to the home platform by computing $f(x)$. However, the home platform doesn't want the target platform to learn anything about the function $f$. The home platform launches the operation by encrypting the function $f$ to get $E(f)$, and then it implements $E(f)$ using the program $P(E(f))$. The home platform embeds the program $P(E(f))$ within the mobile agent and sends it to the target platform for execution. The target platform receives the agent and runs it. This includes executing $P(E(f))$ at $x$ to produce $P(E(f))(x)$. Then, the target platform sends the agent back to its home platform. The home platform extracts the result from the agent and then decrypts it to get $f(x)$.

This solution enables the owner of the agent to execute encrypted programs over untrusted platforms. The executing platforms do not need to decrypt programs before running them. Assume that $f$ is an encryption algorithm or a signature algorithm that contains an embedded key within it. That means that the agent has the ability to encrypt information or sign it without revealing anything about the value of the key being used.

The main challenge in this technique is to find a way to apply it to an arbitrary function $f$. At the moment the only classes of functions for which a suitable encryption is known are polynomial and rational functions [36,38]. Although this technique protects the mobile agent's integrity and privacy, it is vulnerable to certain attacks such as denial of service and replay attacks [36].

## 4.5 Obfuscated Code

*Obfuscation* is a technique in which the mobile code producer enforces the security policy by applying a behavior-preserving transformation to the code before it sends it to run on different platforms that are trusted to various degrees [39]. Obfuscation aims to protect the code from being analysed and understood by the host. Consequently, the host should not be able to modify the mobile code's behavior or expose sensitive information that is hidden inside the code such as a secret key, credit card number, or bidding limits [39].

Typically, the transformation procedure that is used to generate the obfuscated code aims to make the obfuscated code very hard to understand or analyse by malicious parties. There are different useful obfuscating transformations [40,43,44]. Layout Obfuscation tries to remove or modify some information in the code, such as comments and debugging information, without affecting the executable part of the code. Data Obfuscation concentrates on obfuscating the data and data structures in the code without modifying the code itself. Control Obfuscation tries to alter the control flow in the code without modifying the computing part of the code. Preventive

Obfuscation concentrates on protecting the code from decompilators and debuggers.

Hohl [41] suggested using the Obfuscation technique to obtain a time-limited black box agent that can be executed safely on a malicious platform for a certain period of time but not forever. D'Anna et al [39] pointed out that Obfuscation could delay, but not prevent the attacks on agent via reverse engineering. They also argue that an attacker with enough computational resources, such as enough time, can always deobfuscate the code. Barak et al [42] studied the theoretical limits of Obfuscation techniques and showed that in general achieving completely secure Obfuscation is impossible.

In addition to protecting a mobile agent, Obfuscation can also be used for other applications such as protecting digital watermarking, enforcement of software licensing, and protecting protocols from spoofing [39,40]. As far as the performance is concerned, some Obfuscation techniques reduce the size of the code and thus speed up its execution (Layout and Data Obfuscation), while others achieve the opposite (Control Obfuscation) [43]. Obfuscation is considered resistant to impersonation and denial of service attacks [40]. The main challenge in this technique is to make it easy to apply in practice.

## 4.6 Partial Result Encapsulation

*Partial Result Encapsulation* (PRE) is a detection technique that aims to discover any possible security breaches on an agent during its execution at different platforms. PRE is used to encapsulate the results of agent execution at each visited platform in its travel path. The encapsulated information is later used to verify that the agent was not attacked by a malicious platform. The verification process can be done when the agent returns to its home platform or at certain intermediate points in its itinerary.

The PRE technique has different implementations. In certain scenarios, the agent itself performs the encapsulation, while in others the platform does it. To meet certain security requirements such as integrity, accountability, and privacy of the agent, PRE makes use of different cryptographic primitives, such as encryption, digital signatures, authentication codes, and hash functions.

To ensure the confidentiality of its results, the agent encrypts the results by using the public key of its originator to produce small pieces of ciphertext that are decrypted later at the agent's home platform using the corresponding private key. This is one scenario of PRE where the agent itself does the encapsulation process. The agent uses a special implementation of encryption called "Sliding Encryption" that was suggested by Young and Yung [45]. *Sliding Encryption* encrypts small amounts of data within a larger block and thus obtains small pieces of ciphertext. Sliding Encryption

is particularly suitable for certain application where storage space is valuable such as smartcards [46].

Yee [47] suggested "Partial Result Authentication Code" (PRAC), where again the agent does the encapsulation of the results. However, the agent's originator also takes part in this scenario by providing the agent with a list of secret keys before launching it. For each visited platform in an agent's itinerary, there is an associated secret key. When an agent finishes an execution at a certain platform in its itinerary, it summarizes the results of its execution in a message for the home platform, which could be sent either immediately or later. In order to produce the PRAC, the agent uses the associated secret key for the current platform to compute a Message Authentication Code (MAC), which is encapsulated together with the message to produce PRAC. It is important to note that the agent erases the used secret key of the current visited platform before its migration to the next platform. Destroying the secret key ensures the "forward integrity" of the encapsulation results. *Forward integrity* [47] guarantees that no platform to be visited in the future is able to modify any results from the previously visited platforms, as there is no secret key to compute the PRAC for these results. Only the agent's originator has a copy of all used secret keys and thus can verify the encapsulated results. The result verification enables the originator to detect any modification (tampering) of the agent's results. Yee [47] suggested that the results could also be encrypted using the originator's public key, in order to guarantee both privacy and integrity.

Karjoth et al [48] proposed a "strong forward integrity", which, in addition to forward integrity, also requires that the visited platform cannot later modify its own results. Karjoth et al's approach depends on the visited platform doing the encapsulation process instead of the agent doing it. The visited platform encrypts the agent's results by using the originator's public key to ensure the confidentiality of the results. Then the visited platform uses its private key to digitally sign the encrypted results together with a hash chain. The hash chain links the results from the previous platform with the identity of the next platform to be visited. This prevents the platform from changing its results later and thus ensures strong forward integrity [48].

## 5. CONCLUSION

The mobile agent system is a very promising paradigm that has already established its presence in many applications including e-commerce and distributed information search and retrieval. At the same time, this technology has introduced some very serious security problems and emphasized some existing security issues. It is more difficult to ensure

security in the mobile agent paradigm than in some other technologies where hardware solutions are practical.

In this paper we surveyed the main issues in the security of mobile agents. We considered both the mobile agent and the agent platform points of view, and reconfirmed that it is much more difficult to ensure the security of mobile agents than the security of agent platforms. We discussed the security threats and requirements that need to be met in order to alleviate those threats.

We presented the most important techniques for providing security in mobile agent systems. Some of those techniques, for example Sandboxing, have been used for a long time and are well understood. On the other hand, some other techniques, such as Computing with Encrypted Function are still at the theoretical level and are not yet widely used in practice. None of the existing techniques provides an optimal solution for all scenarios. For example, Sandboxing provides a high level of security but is overly restrictive as only a very few applications can operate in such a constrained environment. However, a combination of various techniques may yield powerful solutions. For example, in Java 2 Sandboxing has been used in combination with fine-grained access control and Code Signing. In any case, more research is needed in order to warrant sufficient trust in mobile agent technology by a wide range of users.

## REFERENCES

[1] J. White, "Mobile Agents White Paper," General Magic Inc., 1996.

[2] N. Karnik, "Security in Mobile Agent systems," Ph.D. Dissertation, Department of Computer Science, University of Minnesota, Oct. 1998.

[3] S. Fischmeister, "Building Secure Mobile Agents: The Supervisor-Worker Framework," Diploma Thesis, Technical University of Vienna, Feb. 2000.

[4] W. Jansen and T. Karygiannis, "Mobile Agent Security," NIST Special Publication 800-19, National Institute of Standard and Technology, 2000.

[5] S. McGrath, D. Chac n, and K. Whitebread, "Intelligent Mobile Agents in Military Command and Control," Advanced Technology Laboratories, New Jersey.

[6] G. P. Picco, "Mobile Agents: An Introduction", Journal of Microprocessors and Microsystems, (25):65, 2001.

[7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203--216, Dec. 1993.

[8] D. Rubin and D. E. Geer, "Mobile code security," IEEE Internet Computing, 1998.

[9] D. Chess, J. Morar, "Is Java still secure?," IBM T.J. Watson Research Center, NY, 1998.

[10] L. Gong, "Java Security Architecture (JDK1.2)," Technical Report, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A, 1998.

[11] Li Gong,"Secure java class loading," IEEE Internet Computing, pages 56-61, 1998.

[12] M. Hauswirth, C. Kerer, and R. Kurmanowytsch, "A secure execution framework for Java," In Proceedings of the 7th ACM conference on computer and communications security (CCS 2000), pages 43--52, Athens, Greece, Nov. 2000.

[13] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Dec. 1997.

[14] "Signed Code," (n.d.). Retrieved December 15, 2003, from James Madison University, IT Technical Services Web site: http://www.jmu.edu/computing/info-security/engineering/issues/signedcode.shtml

[15] "Introduction to Code Signing," (n.d.). Retrieved December 15, 2003, from Microsoft Corporation, Microsoft Developer Network (MSDN) Web site: http://msdn.microsoft .com/library/default.asp?url=/workshop/security/authcode/intro_authenticode.asp

[16] Gary McGraw and Edward Felten (1996-9). *Securing JAVA* [Electronic version]. John Wiley and Sons. http://www.securingjava.com/

[17] M. Dageforde. (n.d.). "Security Features Overview," Retrieved December 21, 2003, from Sun Microsystems, Inc. The JavaTM Tutorial Web site: http://java.sun.com /docs/books/tutorial/security1.2 /overview/

[18] R. Levin (1998). "Security Grows Up: The Java 2 Platform," Retrieved December 21, 2003, from Sun Microsystems, Inc. Sun Developer Network (SDN) Web site: http://java.sun.com/features/1998/11/jdk.security.html

[19] P. Lee and G. Necula, "Research on Proof-Carrying Code on Mobile-Code Security," In Proceedings of the Workshop on Foundations of Mobile Code Security, 1997.

[20] A. Appel, "Foundational proof-carrying code," In Proceedings of the 16th Annual Symposium on Logic in Computer Science, pages 247-256. IEEE Computer Society Press, 2001.

[21] S. Loureiro, R. Molva, and Y. Roudier, "Mobile Code Security," Institut Eurecom, 2001.

[22] P. Lee. (n.d.), "Proof-carrying code," Retrieved December 28, 2003, from Web site: http://www-2.cs.cmu.edu/~petel/papers/pcc/pcc.html

[23] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline, "A Certifying Compiler for Java," SIGPLAN Conference on Programming Language Design and Implementation. Vancouver B.C., 2000.

[24] V. Swarup, "Trust Appraisal and Secure Routing of Mobile Agents," DARPA Workshop on Foundations for Secure Mobile Code, Monterey, CA, USA, March 1997. Position Paper.

[25] W. M. Farmer, J. D. Guttman, and V. Swarup, "Security for mobile agents: Authentication and state appraisal," In Proceedings of the European Symposium on Research in Computer Security (ESORICS), pages 118--130, Sep. 1996.

[26] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris and G. Tsudik, "Itinerant Agents for Mobile Computing," Technical Report, Oct. 1995, IBM T.J. Watson Research Center, NY.

[27] J. J. Ordille, "When Agents Roam, who Can You Trust?," Proceedings of the First Conference on Emerging Technologies and Applications in Communications, Portland, Oregon, May 1996.

[28] V. Roth, "Secure Recording of Itineraries Through Cooperating Agents," Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, pages 147-154, INRIA, France, 1998.

[29] V. Roth, "Mutual protection of cooperating agents," In Secure Internet Programming: Security Issues for Mobile and Distributed Objects. J. Vitek and C. Jensen (Eds.), Springer Verlag, 1999.

[30] Y. Ye and X. Yi, "Coalition Signature Scheme in Multi-agent Systems," 2002.

[31] G. Vigna, "Cryptographic Traces for Mobile Agents," in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1419, 1998, Springer, pages 137-153.

[32] H. K. Tan and L. Moreau, "Extending Execution Tracing for Mobile Code Security," In K. Fischer and D. Hutter (Eds.), Proceedings of Second International Workshop on Security of Mobile MultiAgent Systems (SEMAS'2002), pages 51-59, Bologna, Italy.2002.

[33] H. K. Tan, L. Moreau, D. Cruickshank, and D. De Roure, "Certificates for Mobile Code Security," In Proceedings of The 17th ACM Symposium on Applied Computing (SAC'2002) --- Track on Agents, Interactions, Mobility and Systems, pages 76. 2002.

[34] J. Riordan and B. Schneier, "Environmental Key Generation Towards Clueless Agents," G. Vinga (Ed.), Mobile Agents and Security, Springer-Verlag, Lecture Notes in Computer Science No. 1419, 1998.

[35] C. Tschudin, "Apoptosis - the programmed death of distributed services," In Secure Internet Programming [14].

[36] T. Sander and C. Tschudin, "Protecting Mobile Agents Against Malicious Hosts," in G. Vinga (Ed.), Mobile Agents and Security, SpringerVerlag, Lecture Notes in Computer Science No. 1419, 1998.

[37] T. Sander and C. Tschudin, "Towards Mobile Cryptography," IEEE Symposium on Security and Privacy, pages 215-224, May 1998.

[38] M. Abadi and J. Feigenbau, "Secure circuit evaluation: a protocol based on hiding information from an oracle," Journal of Cryptology, vol. 2, 1990.

[39] L. D'Anna, B. Matt, A. Reisse, T. Van Vleck, S. Schwab, and P. LeBlanc, "Self-Protecting Mobile Agents Obfuscation Report," Report #03-015, Network Associates Laboratories, June 2003.

[40] G. Wroblewski, "General Method of Program Code Obfuscation," PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002, (under final revision).

[41] F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts," To appear in Mobile Agents and Security Book edited by Giovanni Vigna, published by Springer Verlag 1998.

[42] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs," in Advances in Cryptology, Proceedings of Crypto'2001, Lecture Notes in Computer Science, Vol. 2139, pages 1-18.

[43] G. Hachez, "A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards," Universite Catholique de Louvain, 2003.

[44] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[45] A. Young and M. Yung, "Encryption Tools for Mobile Agents: Sliding Encryption," In: E. BIHAM (ed), Fast Software Encryption. Lecture Notes in Computer Science, no. 1267. Springer-Verlag, Germany, 1997.

[46] G. Karjoth and J. Posegga, "Mobile Agents and Telcos' Nightmares," *Annales des Télécommunications* Vol. 55, No. 7/8, 29-41, 2000.

[47] B. Yee, "A Sanctuary for Mobile Agents," DARPA Workshop on Foundations for Secure Mobile Code, Feb. 1997.

[48] G. Karjoth, N. Asokan, and C. Glc, "Protecting the Computation Results of Free-Roaming Agents," Second International Workshop on Mobile Agents, Stuttgart, Germany, Sep. 1998.