Countering Agent Security Vulnerabilities using an Extended SENSE Schema

John Page, Arkady Zaslavsky, Maria Indrawan
School of Computer Science & Software Engineering, Monash University, Melbourne
{john.page,arkady.zaslavsky,maria.indrawan}@infotech.monash.edu.au

Abstract

This paper discusses issues involving an intelligent self-reliant mechanism for mobile agents to detect code-tampering attacks made by malicious entities. While describing the security schema, the paper details some scenarios that make the agent vulnerable to malicious attacks. The paper also proposes an extension to the original schema enabling it to cover another security vulnerability in agent operation.

1. Introduction

The popularity and advantages offered by programming languages like Java [1] has encouraged the growth of the agent paradigm. Different studies have attempted to capture the essence behind the motivation for agent technology. One of the most detailed and analytical approaches towards describing the agent paradigm has been done by [8]. While mobility is not an inherent factor in every agent system toolkit, its applicability provides an agent the possibility of extending its function to remote locations. This allows agent developers to maintain a high level of pervasiveness in their applications.

Currently, there are several vendors and research organizations, which have come up with their mobile agent toolkits. Every MAS has its own advantages and disadvantages and is characterized by the services it offers. While some agent systems are suitable for individual agent based application development, others may center on multi-agent applications. Many of these agent system toolkits are used for the commercial implementation of web-based applications. While the growth and the increased acceptance of the mobile agents augurs well for the paradigm, it has also raised several questions relating to security and trust within the agent space.

Security implementations in various systems are designed to allow the agent as well as the agent system to function with a pre-defined level of confidence and trust. However the dynamic structure of the business scenarios, which agents attempt to capture in their function often exposes them to security vulnerabilities. This paper describes an extension of the SENSE schema. The SENSE schema [7] was proposed as a self-reliant security schema for agent execution. The extended version of the schema proposed in this paper, allows agents to perform intelligently with a high level of proactiveness with regards to countering the vulnerabilities described in the paper. To give the reader a perspective of existing security mechanisms implemented in mobile agent toolkits, the paper describes the security implementations of Grasshopper [2] and Tryllian [3]. These agent toolkits have been used as a test-bed for the experiments carried out and for implementing the solutions proposed.

The rest of the paper is organized as follows. The next section, ‘Security in a Mobile Agent System’, describes the security implementations of mobile agent systems, with a focus on Grasshopper and Tryllian mobile agent toolkits. The section, ‘Security Vulnerabilities’, describes various security vulnerabilities that arise from different agent operation scenarios. It also explains how the existing security features are inadequate to counter the vulnerabilities. The next section, ‘The SENSE Security Schema’ briefly describes the SENSE schema and its effectiveness in countering the described security vulnerabilities. The fifth section, ‘Intelligent Countermeasures’, describes a new vulnerability, which can compromise the operation of an agent. To counter this vulnerability, the paper proposes an ‘intelligent’ extension of the SENSE schema. ‘Conclusions and Ongoing Work’, concludes the paper with a summary and an indication of ongoing work.

2. Security in a Mobile Agent System

Most security features implemented in mobile agent systems are designed to function in layers. At the first layer, they attempt to authenticate the agents executing at a particular node. By authentication, the implication is that the agent system attempts to recognize the creator and/or the sender of the agent [4]. Authentication of agents at agent systems is generally carried out using the concept of digital signatures and certificates. The second level of security attempts to check the authorization of the agent. The authorization check is necessary to validate the rights
of an agent at a particular agent system [4]. For example, if an agent indicates that it will attempt to read a file from the local machine, the agent system has to decide if the agent should be allowed to proceed or be blocked. From the implementation point of view, the authorization layer is closely coupled with the authentication layer.

Policy files are used to implement access control in most agent systems. The generation of these policy files is tied in with the authentication layer. For example, if a particular user by the name of developer1 has signed the code of particular agent, the policy file will contain a reference to him or her and accordingly assign access permissions to the arriving agent. Based on a generic security model followed by most agent systems, figure 1 describes the various steps involved in the security processing of an approaching agent. The first step involves authenticating the agent based on the digital certificate carried by it.

The certificate is used to identify the owner or the sender of the agent. If the authentication step is successful, the agent is subjected to an authorization check. The authorization check is used to gauge the privileges for the agent while executing at the server and is made on the basis of the policy files maintained by the agent system. If a valid match for the agent sender is found within the policy file, the appropriate role matching the agent sender entry is assigned to the agent. Every role carries with it certain privileges. It is important to note that some roles may be more privileged than others. In other words, based on prior decisions and role assignments made in the policy file, certain agents may enjoy more freedom than others. While it may be necessary for agents to have a higher degree of freedom, malicious agents may exploit this freedom and use it to inflict damage on the system. The next section describes business scenarios under which the operation of agents can be compromised by malicious entities.

![Figure 1. Agency Level Security Checks](image)

To further understand the concept described above, the next few sub-sections describe the security implementation of two Java based agent systems, Grasshopper and Tryllian. While both these systems, incorporate mobility as an integral feature of their framework; Tryllian has several features to support the development of agent communities and focuses on multi-agent application development.

### 2.1 Grasshopper MAS Security

The Grasshopper MAS has been developed by IKV, Technologies AG. The latest release of the MAS was Grasshopper Version 2.2.4, released in January 2003. Grasshopper security has been built on two levels, referred to as External and Internal levels [2]. The external part of security comprises of the use of socketssl or rmissl. This feature provides a layer of protection for agents when they move between agencies or interact with remote components. The internal part of Grasshopper’s security is based on JDK1.2’s security model. It enforces an access control policy on agents and restricts their rights to perform certain actions such as accessing and modifying files stored on the hard disk. The access control policy is enforced as an access control list, which holds entries for agents. This list is initialized at the start up of the agency. Any modifications to the access control policy can only be affected by the agency.

### 2.2 Tryllian MAS Security

Tryllian is a multi-agent system developed by Tryllian Solutions B.V. Recently; Tryllian released a new version 3.0 of the MAS. The Tryllian MAS is referred to, as the Agent Development Kit (ADK). Since the Tryllian ADK is a multi-agent toolkit, it provides enhanced facilities for agent based collaboration and messaging. Security in the
toolkit is conceptually quite similar to the Grasshopper model of security with a few essential differences. While Tryllian supports X509 certification, TLS /SSLv3, it also maintains an access control policy as in Grasshopper. The essential difference is that it is possible to specify access control based on the agent per developer level. In other words, all agents signed by developer Sam, will be assigned a set of permissions as specified in the access control policy. This policy is controlled by roles. The agent system recognizes four roles. They are All, Basic, NoCert, and Nothing.

Comparatively Tryllian has a flexible architecture with regard to its access control policy. It is also easier to manage and configure the security policy of Tryllian than that of Grasshopper. Tryllian also provides a graphical user interface for the Java keytool to create and manage certificates. The next section examines the security features of these toolkits in greater detail and describes scenarios under which the agent system as well as an agent could become vulnerable to a malicious attack.

3. Security Vulnerabilities

The issue of agent security has been the subject of several discussions and several papers have attempted to tackle the problem. From any standpoint, there are four facets to the question of agent system security [4].

1. Agents attacking Platforms
2. Agents attacking Agents
3. Platforms attacking Agents
4. Agents being attacked by external entities

The first scenario involves the interaction between an agent platform and an approaching agent. Figure 1; in the previous section attempted to diagrammatically explain the steps involved in ensuring the security of the agent system. While authentication and authorization of an approaching agent attempts to protect the agent system, it cannot guarantee that the intentions of the agent will not be contrary to the server interests. In other words the security steps are able to establish the sender of the agent, but are unable to predict the outcome of the agent processing. It is possible that the signature carried by the agent is forged or stolen and on execution, the agent may damage system files, steal data and disrupt the working of the server. In our experiments, it was possible to bring down an agency by making a system call to the exit function. Since the agency had no check in place to block this call, the server exits without any warning and all agents, executing at the server are terminated.

Another disadvantage with code signing is that only static code can be signed [5]. For example, certain parts of the agent code or data, such as itinerary information may be subject to modification while the agent is mobile. Servers may need to be added and some may need to be removed.

Another possible scenario may arise if the agent wishes to modify its self to perform a particular function differently at a particular server. In our security experiments, carried out, using a signed agent it was possible to implement a Denial of Service (DOS) attack on an agent system and affect a general degradation of service. Various manifestations of the DOS attack with screen shots of the attacked agent system have been described in [6]. Thus, code signing is inadequate to guarantee the safety of the server.

In the second scenario, agents attack agents. The motivation behind an attack can be to steal agent code and / or to disrupt the agent operation at the server. Our experiments conducted with the Grasshopper MAS revealed that there is very little protection for an agent against another agent. Any agent entering the agency can access information about other executing agents. This can have far reaching consequences for an agent’s security and privacy. In one implemented attack, the malicious agent was able to read the code of another agent, store it as a data structure within itself and serialize to another server with the stolen agent code. In an extension of the same attack, an agent was able to corrupt the code of another agent by adding junk to its class file. Thus, the agent serializes and moves to another server, it is unable to resurrect itself because of the tampering to its code. A far more sophisticated attack with deeper implications, involved an agent changing the access specifiers of the methods and variables of another agent. This attack makes it possible for an agent to sniff another agent’s private data and methods. Figure 2, is a code snippet illustrating some private data variables.

```
import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
public class AgentBinaryScanner extends de.ikv.grasshopper.agent.MobileAgent {
  /*----------Business Function Statements---------*/
  String agentName,agentId,agentType,agentHome;
  private String name = "John";
  private String address = "Unit 2X/158,Oakleigh Road,Carnegie";
  private String creditcard = "1984827482090";
  /*----------Business Function Statements---------*/
```

**Figure 2. Agent Code Snippet**

Consider the scenario described in figure 3. The screenshot depicts a Grasshopper agency, identified as Destiny Travels. Agents from various agent servers approach the agency for conducting their business transactions. They are indicated by the green figures in the screenshot. The red tip, near at the top of each figure
indicates that they have been digitally signed by their sender or by their creator. This scenario involves the agent identified as John’s Secret Agent. This agent’s genetic construction is similar to the code snippet described in figure 2.

On being granted access to the system, the agent locates all agent class files in the current directory present in its working directory. After it obtains a list of all class files, it inspects the construction of each class file and initiates its malicious action by modifying the access specifiers of all methods and variables marked protected and private. These are changed to public as shown in the agency console screenshot of figure 4. The class file’s length is then adjusted to ensure that the agent is able to proceed with its execution without detecting any changes in its makeup. As a consequence of this malicious action, the agent’s confidential data is exposed to the outside world.

The third aspect to the agent system security arises from an agent system attempting to cause malicious harm on an agent. These attacks are similar to the agent vs. agent attacks documented above. Different countermeasures for agent security have been proposed. [9] presents a summary of these implementations. Agent toolkit designers have dealt with this aspect of agent security in different ways as evident from the previous section, which highlighted some of the security features implemented by Grasshopper and Tryllian MAS.

While these features may be useful in providing a basic level of security, their dependence on cryptographic algorithms and access control lists make them inadequate to meet the changing requirements of different agent operating scenarios. For example it may be possible that an agent enjoying full access privileges docks at the agent system. A lesser-privileged agent can target this particular agent and use it to inflict damage on the agent system. The agent can also steal confidential system information and since it uses a proxy to act, its actions will be undetected by the agent system. Thus agent system security requires elaborate protection measures to guard against malicious agents.

The scenarios listed above highlighted some of the vulnerabilities faced by agents from malicious agents. The fourth aspect of the security question of agents is raised by external entities such as viruses. These external entities interfere with the performance of the agent. To deal with these situations, the agent system employs a secondary protection layer that may involve third party support such as virus scanning software. The next section describes the SENSE security schema [7] and describes its applicability in meeting the security flaws documented. Further the
section also examines another security vulnerability. To counter this vulnerability, the section describes an extension of the SENSE schema, which allows the agent to be proactively monitored from a remote location.

4. The SENSE Security Schema

The SENSE security schema makes an agent capable enough to intelligently handle its security function. The primary objective of the schema is to make the agent self-reliant by enabling it to reliably detect code tampering within its class file structure. Self-reliance implies non-dependence on an external third-party for execution support. It also implies minimal server intervention required in processing the security function.

By using this schema, the possibilities of forged or stolen signatures are also defeated, as the schema doesn’t require the verification of signatures for its execution. Another advantage of the schema is that the agent can apply the SENSE schema over existing security levels. In other words, the agent can still use cryptographic routines and answer to access control policies independent of the SENSE schema. The central idea behind the schema is that the agent is able to scan its own serialized code and compare it to the code-map carried by the agent within itself, as a data structure from its home base. This scan and verification can be done on a periodic basis. A multi-threaded implementation makes the schema intelligent as it enables the agent to execute its security function independent of the original chain of processing. The pseudo code algorithm, in figure 5 describes the original implementation of the SENSE schema.

<table>
<thead>
<tr>
<th>SELF-EXECUTING SECURITY EXAMINATION (SENSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1:</strong> Agent begins execution at the Agency</td>
</tr>
<tr>
<td><strong>Step 2:</strong> Scan Serialized Agent Code</td>
</tr>
<tr>
<td><strong>Step 3:</strong> DO WHILE Agent Processing Incomplete</td>
</tr>
<tr>
<td>If (Agent_Processing is Paused) or (Scan_time_period_flag is TRUE)</td>
</tr>
<tr>
<td>Execute SENSE scan</td>
</tr>
<tr>
<td>If MISMATCH detected</td>
</tr>
<tr>
<td>Agent_SELF_DESTRUCT</td>
</tr>
<tr>
<td>EndIF</td>
</tr>
<tr>
<td>EndIF</td>
</tr>
<tr>
<td>EndDo</td>
</tr>
</tbody>
</table>

**Figure 5. The SENSE Algorithm**

The three steps of the algorithm capture the essence behind the SENSE schema’s execution. The agent scans the serialized code dumped at the local agency and compares it against the code map it carries. The code map is a binary image of its business function.

By executing the SENSE security schema, *John’s Secret Agent* is able to detect a tampering attack made on its code. The sensitivity of the schema can be controlled to detect even minor modifications by setting the precision parameters for the numeric comparisons made within the schema. The screenshot in figure 6 displays a report that describes the salient features of the agent. Apart from its personal details, it also gives the total number of elements scanned and the time taken (in milliseconds) for the scan. In this particular case, the screenshot reports the scanning of a total of 1080 elements. This number varies depending upon the size of the code map considered and the precision settings of the schema.

**Figure 6. A SENSE Scan Result**

The graph in figure 7 gives a performance analysis of the SENSE schema by comparing the run times of two cases. The first series involves the case when the schema detects no tampering and the second series considers the case when the agent code has been tampered with. In all a total of 40 runs were made on *John’s Secret Agent*. Each case involved 20 runs. The selection of a particular case was randomly chosen. Further the scanning initiation of the schema was also initiated with an arbitrary frequency. From the graph it is evident that the processing time taken for the schema operation was relatively uniform in either case. A performance analysis involving variable precision runs has been done in [7].

**Figure 7. SENSE Performance Runs**

This schema performed reliably in detecting a tamper attack made on the class files of an agent. While the SENSE schema can reliably detect a code tampering attack,
there might be other scenarios, which do not involve code tampering, yet their application may render the victim inoperable. The next section describes this scenario and proposes an extension of the SENSE schema to counter this vulnerability.

5. Intelligent Countermeasures

As described in the previous sections, agents can be vulnerable in several ways and agent security should not be limited in its scope of operation. In today’s pervasive world, agents need to proactively consider the consequences of the various interactions that they are involved in. They need to be intelligent with respect to their security.

While the paper considered the tampering of agent code and described the SENSE schema that is able to reliably detect any such attack, there is another dimension of vulnerability that arises from the threaded nature of agents. For example, if an external entity is able to get control to the handle of an executing thread, it can manipulate the status of the same. Thus, agents can be rendered inoperable by external entities such as other agents and agent servers. In the Grasshopper agent system, such manipulation is possible.

Consider the scenario under which agents from different agencies are docked at the Destiny Travels agency. Malicious agents can query the agent system for a list of agents docked at that particular instant of time. Once this list is obtained, the agent can go through the list and suspend the action of all executing agents at the agency. Since the suspended agent is unable to resume execution without external intervention, it is unable to perform any useful action. This scenario highlights the need for agents to handle their security functions proactively. While it would be a trivial solution to block off any external access to an agent’s thread it might disturb the working of certain business scenarios. For example interactions involving agent communities may require agents to suspend and resume each other. Thus blocking every access of an agent will not serve the purpose.

A possible solution to the problem could be an intelligent mechanism that is able to identify valid accesses on the agent. However as highlighted before the agent space can involve the operation of malicious agents with forged or stolen identities. In such cases, it becomes very difficult to authenticate the credibility of an agent each time; it wishes to perform an action. Even if an authentication mechanism were used to do so, the cost effectiveness of the model would severely restrict its viability. For example if a business scenario required an agent, John to suspend an agent, Sarah every hour, authenticating the credentials of John, might appear to be viable but say, for example that there were 500 such agents requiring a similar action every 25 minutes, then the efficiency of this approach would be subject to question.

A workable solution to the problem is the use of a partial Buddy model of agent security [10,11]. In this scenario, every agent, that requires to be suspended, as a part of its operation can delegate a buddy for itself. This buddy agent would periodically monitor the status of its assigned agent from a different agency. If the agent has been suspended, it would investigate the action and accordingly, resume the agent after an acceptable time-period of suspension, as defined for the agent. The architecture of the system is described in figure 8.

Figure 8. An Intelligent SENSE Schema

The pseudo-code in figure 9 describes the working of the extended SENSE schema. It highlights the changed conditions.

| Step 1: Execute SENSE schema |
| Step 2: Decide SUSPENSION Time Period, TP |
| Step 3: Feed Buddy with TP |
| Step 4: Monitor the status of assigned agent. |
| Step 5: If agent status is Suspended |
| Check Suspending Entities Identity |
| Check Elapsed Time period of Suspension |
| Step 6: If agent suspension is Invalid |
| Resume Agent Immediately |
| Else |
| Go to Step 4 |

Figure 9. The Intelligent SENSE Algorithm

Thus this proposed extension allows an agent to proactively monitor its own status, by using an entity external to its operation. Since the agent does not carry any information about its buddy, an external entity cannot deduce the helper’s identification and extend the attack to it. Further each buddy agent in the community is monitored by other buddy agents delegated to it [10,11]. This chain reaction approach allows each agent in the community to participate proactively in the security function while being protected itself. Since the buddy agents may operate from different agencies they will be beyond the sphere of the
malicious action, if a particular agency is malicious. Thus the intelligent extension to the SENSE schema allows an agent to incorporate an increased level of awareness and intelligence in its security operation. This enables it to repel this indirect malicious attack.

6. Conclusion and Ongoing Work

The new generation of MAS based applications are getting relatively more complex and increasingly commercialized. Thus the need for maintaining security in a dynamic and fast changing environment has put several demands on the development of these applications. Most of these demands call for cost-effectiveness, confidentiality and efficiency. Answering these calls requires the application to have a strong backbone of security. Security has become an exponential function of the degree of mobility possessed by the Agent. Some of the various applications that require security are e-commerce based, others are network centric and are focused on answering network management issues like reducing the network load, overcoming network latency and executing asynchronously, while maintaining an acceptable level of autonomy. In this paper we analyzed the security aspects of mobile agents operating under vulnerable scenarios. The paper described the implementation of the SENSE schema in countering these vulnerabilities.

The SENSE implementation is an attempt to make protection and security an inherent functionality of the MA operation. It equips the MA with ability to self-scan itself at random intervals, making a subversion attack almost impossible. A performance analysis describing the uniformity in execution time under different scenarios was also given. Further the paper described a new vulnerability that compromises the agent action. To counter this vulnerability, an extension of the SENSE schema was proposed. This provides the agent with an intelligent approach for countering the vulnerability described. Future work will explore new vulnerable scenarios of different agent systems and extend the SENSE algorithm to counter the vulnerabilities discovered.

References