Evaluating Security in Software Agent Systems using a Security Analysis Tool

John Page
Arkady Zaslavsky
Maria Indrawan

School of Computer Science and Software Engineering
Monash University
e-mail: john.page@csse.monash.edu.au,
e-mail: arkady.zaslavsky@csse.monash.edu.au,
e-mail: aria.indrawan@mail.csse.monash.edu.au

Abstract

This paper exposes different security threats posed to information in software agent systems. It describes various scenarios in which security can be compromised and highlights the requirements of a standard security-testing environment. This environment is generated using a Security Analysis Tool (SAT) proposed and described in this paper. The tool also fills in the requirement of a much-needed benchmarking function for the security implementations of different mobile agent systems.

Key Words

Agent, Mobile Agent (MA), Mobile Agent System (MAS), Attack, Security

INTRODUCTION

Today, we live in a pervasive world, in which Information Systems (IS) are mobile and there is a rapid movement of data and information within them. The need for an IS to have access to a constant stream of reliable data while being mobile has led to the development of Mobile Agent (MA) based systems. The MA paradigm allows the generation of pervasive environments for the dynamic exchange of data and information between nodes. Further, it allows the application users to remain virtually connected while being physically disconnected from the network. The significant advantages in computing cost and power have led to the development of several agent-based applications generating significant interest in agent led computing.

Several viewpoints, explaining the motivation behind this paradigm, have been put forward. Franklin et al (1997), describes an agent as a user’s intention and authority, whose goal is to achieve pre-programmed objective(s) to the best of its ability. Integrating this concept with mobility makes an agent, an intelligent piece of software possessing a unique data state, a definite behaviour and a variable location parameter. Mobility enables the agent to move across different networks and nodes, while attempting to meet its goals and objectives thereby improving its chances of success. Cabri et al (2000) have classified the different degrees of mobility exhibited by MAs into strong and weak mobility. Strong mobility implies the movement of MA code and its current state of execution. Telescript developed by General Magic (White, 1994) implements this level of mobility. On the other hand, weak mobility, which is exhibited in MAS developed using programming languages like Java (Gosling, 1996), do not permit the transfer of execution state. However the advantage of platform independence and portability has allowed Java to overshadow this limitation, and be chosen for the development of several MAS toolkits.

The advantages of the MA paradigm are felt most in web-based applications where bandwidth is of a premium. MAs reduce computational bottlenecks by shifting tasks from heavier loaded nodes to those comparatively less loaded. Another advantage of this mobility is in situations when the reliability of the computing server is not certain. In such scenarios, MAs pack up the current state of the process and transfer it to another available server for further processing. Applications that benefit from this paradigm are real-time conferencing software and multi-user applications. A good example of such a multi user application is computer gaming software, requiring a high amount of graphics support. MAs are also being used in different e-commerce applications. Their use has been envisaged as personal assistant agents, trading agents and data gathering agents to name a few areas of application. Lange (1999) has given seven good reasons for using MAs and currently, many universities and research institutions are working towards refining and extending the applicability of MA based systems.

While
This paradigm presents an interesting facet of pervasive computing, it also gives rise to security related issues, affecting MAs as well as the agent servers.

This paper analyses security threats and vulnerabilities, which arise while using MAs in a distributed environment. Various systems propose different schemes for security and currently; there are no available means to test the proposed schemes. This paper proposes and describes a security analysis tool (SAT), which can be used to probe different security implementations of different agent systems. The paper is structured as follows. This section gave a brief introduction of the MA paradigm and its applicability in an information sensitive scenario. The next section “Attacks on a MAS environment” analyses the security vulnerabilities of this paradigm by describing scenarios in which the working of the MA is compromised. This section also discusses the implications of these vulnerabilities and describes attacks we carried out within a MAS environment using the Grasshopper (2001) v2.2b MAS. The section, “Requirements of a security test environment” examines the building blocks of a standard security-testing environment. This environment can be used to analyse the strength of various security features across different implementations of MAS. This analysis is carried out using a Security Analysis Tool (SAT) introduced in this section. The section, “Working of the security tool” describes the implementation of the SAT in more detail and the final section concludes the paper with an indication of future work.

ATTACKS ON A MAS ENVIRONMENT

Different nodes participate in providing travelling MAs with resources and information. Since the network is a dynamic entity with servers connecting and disconnecting with variable frequency, the availability of secured channels for agent travel becomes an unknown parameter. This unpredictability introduces security threats into the framework of the system, as the non-availability of secure nodes causes the MA to alter its travel itinerary and travel through un-secured channels. Such scenarios allow malicious entities to gain, steal and capture information.

On the basis of their approach, attacks can be categorized as Active and Passive (SSH Communications Security, 2002). Active attacks do not conceal their operation. Their intention is to steal or destroy information being carried by the MA. These attacks attempt to capture and destroy the MA and in some cases, prevent it from returning to its home base. Passive attacks are subtler in nature. The goal behind these attacks is to capture data while preventing the MA or its parent server from becoming aware of these thefts. As obvious, attacks of this nature are more sophisticated. Malicious entities employing a passive attack will attempt to subvert the MA and use it for its own interest.
Figure 1. Different Attack Scenarios in a MAS Space

The different types and variations in attacks can be understood by considering an example within the airline travel industry in which Alibaba, Eureka, Asiana and Maxima are four different travel agencies who offer their services to the general population using a web-interface. Sam is a frequent traveller who employs MAs to mine the Internet for the ‘best’ deals available. Asiana and Maxima have an agreement under which Asiana will direct all travel inquiries related to Europe to Maxima, which in return will forward all business opportunities for travel within Asia to Asiana. Such agreements are understood and recognised by the agent servers, which receive MAs and execute them. In the scenario described in Figure 1, Sam wishes to fly from London to the Bahamas. Accordingly, his MAs are sent out to different travel agencies, including the servers of Alibaba, Asiana, Eureka and Maxima. The agents have been programmed to search for strings matching, “Airline Ticket”, “Economy Class”, “LDN-BHM” and “October 2003”. The MA will filter through the information it encounters and only bring back data matching all four strings at one server. Sam’s MAs can be attacked actively as well as passively. These attacks are described below

Case 1

In case 1, the Alibaba server launches an active attack against the agents. Alibaba is aware of its competitors in the market and keeps a close watch on the agents approaching their servers. Since secured channels are not available for Sam’s agents to travel, they have to use servers, whose credibility has not yet been established. These servers may harbour malicious intentions against other MAS and destroy their agents, when they dock. Destroying an agent is just one of the ways of launching a malicious attack and not a very long-term option for the attacker to exploit. The reason being that once agents do not report back from destinations to which they have been sent, the parent server of the agents becomes aware of a malicious activity from certain IP addresses and these are then blacklisted. Hence, unless the malicious server is not employing IP spoofing, this version of direct attack would not serve a long-term purpose for it.

Attackers are always on the lookout to exploit some weaknesses of the MA framework, on a long-term basis. Therefore, the use of an active attack, in which the identity or the source of the malicious attack is not concealed, is seldom deployed. Another variation of the active attack is the Denial of Service (DOS) attack. This attack is used to prevent legitimate agents from docking on to the agent server by consuming the entire resources of the server. Figure 2, depicts the Grasshopper Agent Server, console while this attack is in progress.
In the figure, the memory status of the system is shown alongside the server log and console, using a Windows Task Manager. With just ONE Grasshopper malicious agent, it took 52 minutes to implement a DOS attack on a Grasshopper Agent Server. The agent server is using J2SDK version 1.4.0. The operating system is Windows 2000 and the memory is 256MB. The malicious MA docks at the agent server and starts replicating itself at the rate of 2 agents every second on an average. Within 6 minutes, it ties up 99% of the CPU. In 20 minutes, it had caused the system to spawn 3977 threads, which do nothing apart from spawning more threads. This attack has been implemented when conditions for the system were favourable. In other words, at this point of time, no other agents were docking at the server and no other processes were requesting memory.

In other variations, it is possible to execute the same DOS attack using a team of agents, which will dock with other valid entities. This will make it difficult for the server administrators to detect the malicious entities easily. Figure 3 depicts the memory usage of the system using the Windows Task Manager after the attack was terminated.
Apart from the DOS attack, malicious agent servers using their own agents target other MAs and agent servers. In the travel industry scenario described before, malicious agents will attempt to intercept and attack agents while docked at neutral servers. For example, it may happen that Alibaba’s agents dock onto the Eureka’s server and attack agents, redirected from the Maxima agent server. In another variation of an active attack, it was possible to bring down the MAS server using a rogue agent. This was possible because the MAS server allowed the MA, the flexibility to execute system commands from within the agent environment.

Case 2

In case 2, the Alibaba server launches passive attacks on other agent servers by sending malicious agents. These attacks attempt to capture data carried by the agent as well as gain insight into its working. This information will enable Alibaba to construct agents resembling the agents of other agencies, and enter into false business deals. For example, in one scenario, Alibaba replicates the agents sent out by Sam and sends them to make bulk travel bookings at Eureka. Since at Eureka, Sam’s MAs have a preferred customer status, the bookings are allowed. Meanwhile, other legitimate MAs approaching Eureka are unable to make bookings as the malicious agents sent out by Alibaba have blocked seats. This not only results in loss of business for Eureka but MAs redirected to Alibaba, may end up purchasing the blocked seats at a higher price. Further Sam, will be billed for seats that he did not book. This black-marketing approach of malicious MAS is one of the ways of implementing a passive attack.

Another variation of a passive attack is the DOS attack, described in the previous case. In this attack, implemented in the same Grasshopper MAS environment, malicious agents from the Alibaba server dock at the Maxima, Eureka and Asiana servers and poll the MAS for a list of agents docked at that particular server. On getting this list they change the status of each agent from ACTIVE to SUSPEND. Since MAs in the Grasshopper environment are unable to change their status back to ACTIVE, without the intervention of the MAS server or an external entity such as another agent, these MAs are unable to continue with their processing. These suspended MAs continue to use system memory without doing any useful processing and because they are blocking resources, legitimate MAs approaching the server are turned away.
In the scenario depicted in figure 4, a malicious MA identified as John’s Suspending Agent docks at the system. On entering the system, it searches for the place named as Sweden as it is specifically targeting agents docking at that place. In the Grasshopper context, the term place presents a logical grouping of services to the agent. In the server log shown in figure 4, the malicious agent lists all other agents at that particular agency and then starts suspending them individually. The console of the MAS, on the extreme left of the figure, indicates two MAs namely Grasshopper Agent and PrintInfoAgent, which have been suspended. A small circle near the head of the green MA icon indicates that these agents are in a suspended state. The DOS passive attack was also launched by sending a malicious MA, which clones itself and each clone, replicates itself. This is similar to the active attack described previously with one essential difference. In this case, the MAs destroy themselves periodically and clean up the memory, allowing the system to recover just before it crashes. In the meantime, other agents wishing to connect to the server get turned away from the MAS. Another passive attack is possible by modifying the data carried by the MAs. Consider the case in which Sam’s MAs dock at Alibaba’s server after passing through the servers of Maxima, Eureka and Asiana. The MA carries a price quote from each of these three servers for the LDN-BHM travel leg. Since all MAS servers have complete control over the MA, entering its environment, the Alibaba server tampers with these prices and transmits the MA with incorrect data back to Sam’s server. Sam will naturally reject prices quoted at a higher rate and consequently deals brought back by agents from other servers will get rejected, even though they were originally better.
REQUIREMENTS OF A SECURITY TEST ENVIRONMENT

Testing is an essential component of the software life cycle. Boehm et al (1978) has given a set of criteria for evaluating a software product but the model is unable to make a clear connection with usability and functionality as pointed out by Sulaiman (2003). While different viewpoints define the stages involved in testing in different ways, Sommerville (2002) defines five stages of testing – unit, module, subsystem, system and acceptance testing. While these stages are important to evaluate the software from a functional point of view, the SAT proposed in this paper attempts to analyse and compare the strength of different security implementations in agent systems. This testing will focus on the relative strength and the applicability of different schemas rather than their functional correctness. The previous section highlighted a few of the security vulnerabilities, which are exploited by malicious entities in a dynamic MAS environment. The need for a configurable test environment to evaluate the security of different MAS servers, implemented across platforms is imperative. This section develops the framework and outlines the requirements of a generic security test environment.

The requirements of this security test environment can be categorized as

1. **Modularity:** The entire framework should be implemented in modules, which can be added, removed and modified to suit the changing requirements of different test conditions. For example, in some cases there may be a need to test the cryptographic capabilities of a security implementation within a particular MAS environment. This scenario may require the generation of test cases specific to the scenario. This cannot be handled if there is a general interface for feeding in the input test parameters. If the test system is designed to be modular at each level, greater flexibility can be achieved while testing the system.

2. **Configurability:** This follows from the above requirement. The test environment needs to be configurable to allow different parameters and different levels of testing. For example, different entities might act upon the system at different points. There might be an interaction between MAs, or between MAs and the MAS. All these entities may have different access permissions within the environment. Even between MAs, there may be different levels of authority. Some agents may work in a supervisory mode and instruct others while this policy may not hold good for others. Creating a configurable test environment allows these situations to be represented.

3. **Traceability:** The test environment needs to reflect the security design goals of the MAS as laid out in the Software Requirement Specification (SRS). The environment should be able to generate a traceability matrix, which can clearly match these goals while analysing the functionality from the test cases. For example, the test cases should be written using formal parameters to reflect specific security features implemented in the MAS.

4. **Benchmarking:** An important but difficult to implement feature in creating generic test environments is the comparison of security features across different implementations and MAS. For example, a MAS might rely on the security protection made available by the underlying execution environment. In Java based MASs like Tracy (Braun et al, 2002), the agent server are protected using the sandbox model of security. MASs like Agent TCL, developed using TCL, a scripting language created by Dr. John Ousterhout; rely on authentication and encryption using PGP (Gray, 1995). The security-testing tool should be able to benchmark the strength of these different methods and offer a comparative analysis.

The security-analysis tool (SAT) proposed in this paper captures the above-mentioned requirements while being generic in nature. The goal is to make the tool intelligent enough to evaluate and compare security features of different MAS using the same scale. Figure 5 explains the design functionality of the SAT. It describes the four dimensions of the tool in terms of the parameters that are accepted by it. The CORE is the central testing algorithm of the tool. It interlinks the parameters received and analyses the results. Since the SAT is designed to work in pervasive scenarios, it has to be robust in order to meet different conditions imposed by changing parameters. For example, in a particular scenario, it might be called upon to test the security policy of an agent server. This security policy might be implemented using access control lists and policy files. In this situation, the tool might require reading these policy files in order to compute and evaluate the validity of the security implementation. Another agent system might use encrypted functions to enforce security and access control between the entities. The tool might be called upon to test the effectiveness of the encryption technique used by these functions. To meet these different scenarios, the CORE is designed to be generic without having too many dependencies. This enables it to evaluate different implementations and set benchmarks against which a comparative analysis can be done.
As shown in figure 5, there are four facets to the SAT, which are defined as follows.

A. **Input parameters:** This refers to the values, which will be made available by the test case during the security testing. Specifications relating to data type, data range and the actual values to be fed in are given here.

B. **Output parameters:** This refers to the expected values, which should be generated by the test case. Specifications relating to data type, data range and the expected values are fed in here.

C. **Storage structures and limits:** This relates to the storage structures and their limits, which might be used by the tool during its internal processing. The idea is to give the tool some indication of the expected processing. The SAT will use this information to generate deviations between expected and actual results.

D. **Access levels:** This feature will attempt to enforce access restrictions on the security feature under test. For example, if the security feature under test is visible as Process A within the environment and the access level has been set to 1, then no other process should be able to stop it from completing its run. In other words, it should have complete priority. The same feature, if tested when the access level is 2 should allow other processes to communicate with it while running.

**WORKING OF THE SECURITY TESTING TOOL**

The CORE uses the four parameters described in the previous section and executes the test case supplied. It then applies various combinations with the data sets supplied to it to analyse the internal processing of the MA code in terms of Processed Instruction Code (PIC) at certain virtual code breakpoints, called Performance Indicator Points (PIP). PIC refers to special variables and data structures, which are placed at key branches within the MA code. These variables and data structures play a critical role during the execution of the code and the values held by them serve as pulse indicators of code health. To understand this, consider the code snippet shown in figure 6. In this snippet, a PIC variable named \( CTR \), is placed within a succession of IF and FOR Loops.

![Figure 5. Design Functionality of the SAT](image-url)

**Figure 5. Design Functionality of the SAT**

If \( \text{divX} = 34 \) then
  Call Proc PrintValue; /* This initialises CTR */
Else
  Call Proc RegenerateGraph; /* This initialises CTR */
  
  For \( i = 1 \) to \( \text{divX} \)
    
    \( \text{term} = \text{func} (\text{modX}); \)
    \( \text{endterm} = \text{term} * 67; \)
    \( CTR = \text{endterm} \mod X \)
End For

**Figure 6. Code Snippet explaining a PIC variable**
The value held by this variable is monitored at different points during the execution. The points at which these values are examined are referred to as Performance Indicator Points (PIP). PIPS function as break points for an analysis run. The volume of instruction code and the complexity of its functionality decide the number of PICs and PIPs to be used.

PIC values are recorded at these points in two different cases.

**Case 1:** The security feature is tested in a stable environment, in which resources are readily made available and NO attacks are launched against the environment. This test run is used to map the internal processing of the MA and to initialise the PICs.

**Case 2:** The security feature is tested in an unstable environment in which resources are withheld and malicious attacks are attempted against the environment. As in the previous test run, the processing of the MA is measured at the pre-defined PIPs.

Variations in the values of PICs between the two test runs are used to analyse the degree of abnormality exhibited by the MA while under attack. The results of this analysis can be used to predict the performance of different security implementations in various scenarios. Table 1 gives a partial sample test case, explaining the working of the SAT. This particular test case defines five PIPs and three PIC variables. The values of these PIC variables are initialised during the first run of the test case, with the Access Level set to 1. This is an ideal scenario under which the functionality is expected to perform optimally. Readings during this run are recorded and compared when the test case is re-run after changing the Access Level parameter.

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case COMMENTS</th>
<th>Test Case ACCESS LEVEL</th>
<th>Test Case INP PAR</th>
<th>Test Case OUT PAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz123</td>
<td>Cryptographic procedures of Tracy MAs to be checked</td>
<td>1 (which means no attacks will be launched)</td>
<td>NUM=32, STR1=“XYZ”, STR2=“RTW”, RANGE= 0</td>
<td>NUM1=447, NUM2=889</td>
</tr>
</tbody>
</table>

**STORAGE STR** *(At a high level)*

<table>
<thead>
<tr>
<th>PIP</th>
<th>PIC 1</th>
<th>PIC 2</th>
<th>PIC 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIP 1</td>
<td>12</td>
<td>“@er”</td>
<td>0.23</td>
</tr>
<tr>
<td>PIP 2</td>
<td>37</td>
<td>“ktr”</td>
<td>0.53</td>
</tr>
<tr>
<td>PIP 3</td>
<td>45</td>
<td>“ter”</td>
<td>0.3</td>
</tr>
<tr>
<td>PIP 4</td>
<td>41</td>
<td>“jer”</td>
<td>0.009</td>
</tr>
<tr>
<td>PIP 5</td>
<td>40</td>
<td>“Ner”</td>
<td>0.09</td>
</tr>
</tbody>
</table>

The Access Level parameter can indicate various other settings apart from priority settings, depending upon the changed environment conditions. In the test case shown in Table 1, the value is set to 1, which implies a STABLE environment. Other values can be 2, which indicate a SEMI-STABLE environment. This implies an environment in which no attacks are launched but the executing MA is not allocated the resources it requires and is kept on hold. A value of 3 can indicate an active DOS made on the agent code, while a value of 4 may indicate a passive DOS. The CORE, which is the testing engine, will use these values to execute a test run on the MA execution. The first test run will always be to verify if the program performs according to expectations. The value of the OUTPUT parameter will be compared with those actually generated. Any deviations will be investigated. Only when there is a complete match in the OUTPUT parameters between expected and actual values, will the security analysis carried out.

An algorithm explaining the different steps of the SAT is as follows.

/*Algorithm of Security Analysis Tool (SAT) Functionality */

/* Pre-Processing Initialisation Stage*/

**Step 1:** Acquire INPUT and OUTPUT parameters.

**Step 2:** Define and Initialise Variables and Storage Structures.

**Step 3:** Define PICs and PIPs in Process Monitor (PMON)

**Step 4:** Set Access Level to STABLE

/* Intermediate Initialisation Stage*/
Step 5: Execute the CORE.

Step 6: Check received output parameters with OUTPUT parameters.

Step 8: if NO deviation /* This implies Program Run was correct */

Store values of PICs

Else/

Discard PICs

Investigate reasons for deviation

Check environment settings

GOTO Step1

/* Final Run*/

Step 9: Set Access Level to UNSTABLE

Step 10: Execute the CORE

Step 11: Acquire values of PICs for this run

Step 12: Generate values of deviations for comparative analysis at different PIPs

The results of the SAT can be used for analysing and evaluating the performance of different security features implemented across various MAS. It can also be used to generate scenarios in which the security of different applications might get compromised. In other words, if a particular analysis of a security implementation reports it to hold good under the applied conditions, methods like extrapolation could be used to predict conditions in which the implementation will fail.

CONCLUSIONS AND FUTURE WORK

Information systems developed using MAS operate in pervasive environments. They require a constant level of information flow and are therefore highly vulnerable to hostile action from malicious entities. This paper analyses the security vulnerabilities of software agents in a dynamic environment. It also categorizes and explains the different kinds of attacks possible. Different MAS used in various web-based applications employ different security implementations for meeting the vulnerabilities that might arise. Currently, there is no available platform for testing these implementations. This paper contributes to the area of security analysis in pervasive applications. It highlights the need for a security-analysis tool (SAT) to evaluate the security implementations of different agent based systems. It enumerates the design objectives of a security-testing environment and maps them to the functionality of the SAT. Another advantage of the SAT is that it be extended to predict scenarios in which security might be compromised.

Future work will concentrate on further refining and extending the design of this tool to expose new security vulnerabilities in agent systems. This will help in the development of new security implementations for MAS based applications. The target will also be to make the SAT flexible for it to be applied directly at the MAS environment where the security feature is to be tested, rather than bringing the MAS to the SAT environment.

REFERENCES


Sommerville, I. (1992), Software Engineering, Addison-Wesley, Wokingham, England


COPYRIGHT

[John Page, Arkady Zaslavsky, Maria Indrawan] © 2003. The author/s assign the We-B Centre & Edith Cowan University a non-exclusive license to use this document for personal use provided that the article is used in full and this copyright statement is reproduced. The authors also grant a non-exclusive license to the We-B Centre & ECU to publish this document in full in the Conference Proceedings. Such documents may be published on the World Wide Web, CD-ROM, in printed form, and on mirror sites on the World Wide Web. Any other usage is prohibited without the express permission of the authors.