Self-Protected Mobile Agents

J. Ametller, S. Robles, J. A. Ortega-Ruiz
Department of Computer Science, Universitat Autònoma de Barcelona
08193 Bellaterra - Spain
Joan.Ametller@uab.es

Abstract

In this paper, we present a new solution for the implementation of flexible protection mechanisms in the context of mobile agent systems, where security problems are currently a major issue. In our scheme, agents protect their code and data by carrying their own protection mechanisms. This approach improves traditional solutions, where protection was managed by the platform. The implementation is far from trivial. We have implemented this scheme in the JADE framework, using Java. Any application using mobile agents can incorporate these mechanisms to implement agent protection with a minimum impact in the its existing code base.

1. Introduction

Mobile agent systems combine the agent and mobile code development paradigms, incorporating to agent platforms the ability to move computations across the nodes of a wide area network. However, mobile code systems raise a well-known set of security issues [5, 7, 2] that have to be addressed by any platform providing free-roaming mobile software agents. These security concerns can be classified in two broad categories, according to whether the agent's or the platform's security is at stake.

On the one hand, host platforms receiving and executing mobile agents must be protected against malicious code. Common mechanisms addressing this issue include cryptographic authentication and integrity checks, code signing and encryption, etc. On the other hand, mobile agents must protect themselves against hosts trying to tamper maliciously with either the code or the data carried by incoming agents. This issue, known as the *malicious host problem* [11, 7], is usually addressed by the introduction of application-level cryptograhic protocols [6, 10, 9, 8] whose aim is providing two basic guarantees: confidentiality and integrity.

Confidentiality issues arise specially in the context of mobile agents carrying data that must be accessible only to specific, authorised hosts in their itinerary. For instance, the hosts in the agent itinerary could be direct competitors as providers of a resource looked for by the agent (e.g. [2] discusses the example of an airlines ticket booking agent.) Besides barring access to reserved information, the roaming agent must also ensure the integrity of the data it carries, i.e., any tampering with pre–existing data must be detected by the agent's owner and, if possible, other hosts in the agent's itinerary.

The aforementioned protocols address the confidentiality and integrity problems with different degrees of success, but, as shown in [10], never in a completely satisfactory way. In any case, all of them are based on standard cryptographic schemes, often relying on public key infrastructures, via platform driven protection mechanisms. Unfortunately, this security solutions often rely on a static prospect, implying the modification of all the involved platforms.

In this paper we present a general software architecture for the protection of mobile agents, with the aim of minimising or even getting rid of some of the main difficulties of existing solutions. Our scheme merges the agent and platform driven approaches into a flexible method for the protection of agent's code and data. Existing cryptographic protocols can be easily embedded in our solution, avoiding in the process some of their shortcomings. The key idea of our approach lies in enhancing agents with an independent, fully encapsulated protection mechanism carried by the agents themselves. This security layer interacts with platforms in very definite circumstances via clear-cut interfaces, minimising the impact (in terms of new developments and legacy code reuse) of adopting the new mechanism or even modifying the underlying security policies and techniques. This is a telling argument of our approach, for the infrastructure of platforms becomes tenable. Users must not learn about security since the mobile agents carry their own self-protection mechanisms, wrapping the application layer. Last but not least, domain-specific or home-brewed security mechanisms can easily coexist with the new architecture.

It is worth stressing that existing agent-based applications can benefit from this solution. Making mobile agents secure involves only minor changes to both the platform and the agent's code. The new ideas can also be applied to other innovative concepts such as self-interpreted code or self-extracting agents.

Finally, we do not limit ourselves to a theoretical speculation on the benefits of our solution. A fully functional proof-of-concept implementation of the work described in this paper (using the well-known JADE agent platform), briefly discussed in section 5, has been developed by the authors.

2. Scenario

As we have outlined in the introduction, mobile agent systems pose numerous security and software engineering challenges, affecting several areas of their architecture, design, implementation and deployment. This paper focuses on the issue of protecting the agent's data and code against malicious or unauthorized entities.

Consider the common scenario of a mobile agent following a (possibly pre-established) itinerary through an heterogeneous set of agent platforms. At each stage of its travel, the agent uses the given platform services together with self-carried code to accomplish its goals. We will be interested in two immediate issues concerning the agent-platform interaction, namely

- Security. It is necessary to protect the mobile agent in terms of privacy and integrity against third-party entities. Although the agent can, in principle, trust the platforms that it visits (other malicious-host problems [4] are out of the scope of this paper), only some parts of the agent code will be used in each platform. In scenarios where direct competition between visited platforms exists, it is often desirable that some parts of the agent's code and data are visible to just one platform, and totally unaccessible the other ones.
- Interoperability. Reliance on platform-specific interfaces impedes transparent agent migration across heterogeneous platforms. Thus, the platform-agent interface should be as simple and generic as possible. Nonetheless, in the absence of widely accepted, standardized interfaces, developing multi-platform agents implies providing adequate means for a clear-cut separation between agent generic behaviors and platform-specific code.

Our security requirements can be attained via public key cryptography. Several proposed schemes [10, 9, 8] already use public key cryptography to address, among others, some form of the above security concerns. Despite their differences, all these schemes share a common underlying scenario. The agent's originator and each of the platforms in its itinerary own an asymmetric key pair. The corresponding public keys are available to any interested party (possibly by means of a PKI [12]) and standard cryptographic mechanisms (like digital signatures and encryption) are used to protect sensitive information. Thus, any data exchange between the traveling agent and the platforms it visits can be protected, either to ensure its integrity (digital signatures) or to avoid disclosure to third parties (encryption). In addition, the use of some form of bytecode allows to apply these security mechanisms also to executable code.

On the other hand, the interoperability requirements are not as easyly solved. Typically, the implementations of the above mentioned schemes provide large, non-trivial agent interfaces to cover all their security requirements. Unfortunately, such overly rich interaction mechanisms force agent implementors to follow a very concrete security scheme, even in cases where other protection mechanisms are required, or where security measures are not needed or desirable at all (e.g., for efficiency reasons).

The rest of this paper presents a scheme to satisfy our security mechanisms without the interoperability shortcuts that affect current proposals. As we will see, these requirements will lead us to a mobile agent architecture based on public key cryptography and aggressive code-as-data representation. A thin, explicit code layer is then in charge of interpreting this data, after it has been adequately checked and decrypted. Here, we must face a serious problem, which in fact challenges the adequacy of the whole proposed solution: how are these cryptographic services provided and used so that the security of the system is not compromised? Let us see in more detail where and how this problem arises.

When the agent arrives at one of its target platforms, the pertinent code and data must be decrypted using the platform's private key. Obviously, there are only two possibilities as to this task's responsible: the decryption process is carried out either by the agent itself or by the platform. Alas, both scenarios are plagued with serious security and interoperability problems:

Platform-driven decryption This is the traditional approach used by most implementations. The agent's data and code to be executed in a given platform is decrypted by the platform itself. This scheme is at odds with our portability requirements, since it constrains the agent itinerary to platforms understanding a substantial part of its internal organisation. Any non-trivial change in the agent's structure will imply modifications on all target platforms. Moreover, supporting several protection schemes enforces the platform to distinguish between different agent interfaces.

Agent–driven decryption Agent-based decryption is probably the best solution from the interoperabil-

ity point of view, in the sense that the interface offered by the platform is minimized: all that is needed is a function giving access to the private key, and platform implementors are not concerned on any detail of the internal agent structure. In this way, not only can agents protected with different cryptographic methods or unprotected agents live in the same MAS, but also new protection schemes can be added without modifying the existing platform code. To perform the decryption activities itself, the agent must gain access to the platform's private key. This option poses obvious security threats: just think of network sniffers, man-in-the-middle attacks and the like. In addition, third-party malicious code injection in the agent could modify its behavior and take profit of its accessing the platform's private key to, for instance, obtain data intended only to the platform at hand.

Thus, we are faced with a seeming conundrum: either we offer a secure execution environment for our mobile code (using platform-driven decryption) and seriously hurt its re-usability and interoperability, or we prime the latter by moving the relevant operations within the agent, but raising in so doing unsurmountable security risks. Our aim in this paper is to offer a way out this dilemma, and to reconcile our apparently contradictory requirements. We will show how a minimal, standards-based decryption interface, combined with an appropriated agent architecture, allows secure mobile code execution, hiding at the same time its details within the agent code, i.e., not requiring the platform to be aware of the agent internals.

3. Architectural overview

This section presents our proposed solution to the interoperatibility and security problems arising in the mobile agent scenarios described in section 2. We present a software architecture that merges the agent and platform driven decryption approaches, fully addressing and solving their discussed shortcomings.

Our solution affects both the agent's and the platform's architecture, but minimises the impact on the latter, fostering its interoperability and ease of adoption by current platform implementations.

As we have seen, the main risk in agent-driven protection mechanisms is that agents need access to the platform's private key. We can circumvent part of this problem by requiring a *cryptographic service* as part of the platform's standard interface. In this way, we avoid direct manipulation of the private key by agents—but we must also avoid that malicious third-parties use this service to decrypt data stolen from itinerant agents. Thus, the cryptograhic service interface will incorporate some sort of verification mechanism to ensure that decrypted data will only be handed to the agent for which it is inteded.

The cryptographic service can be provided as an add-on in existing platforms, via either a software service or an autonomous agent, and in a way that minimises the knowledge of the internal agent structure needed by the platform. To this end, the mobile agents must be internally (re)structured according to the following scheme:

- All executable code intended for a given platform will be wrapped up as data (e.g. as Java bytecode) and encrypted, possibly with accompanying pure data, using the platform's public key.
- The above data will be transported and handled by explicit agent code (C) that will use the platform's cryptographic service to decrypt it. C contains no platform-specific functionality.

That is, mobile agents will be always structured as a pair (C, D), where C is portable, explicit code and D wraps as data the agent's specific (and maybe also platform-specific) behaviour. It is important to stress that this requirement on the agent's architecture in no way precludes reuse of pre-existing agents, since it is easy to make them compliant by standard software engineering design practices such as the *Adapter* pattern [3]. In particular, the data-code split can coexist with other architectures already proposed to improve platfom interoperability.

Having addressed our interoperability concerns, it remains to show how this architecture also ensures the security of the whole scheme. Thus, the following section describes how the agent code C and the cryptographic service collaborate to protect the agent's code and data from external attackers.

4. Protection Mechanisms

4.1. Public decrypting function

The main component of the platform's cryptographic service will be a public decrypting function. More concretely, let $D_j = E_{k_j}(m_j)$ be the result of encrypting the data chunk m_j using k_j, the public key of the jth platform.

As already stated, our goal is to ensure that only the authorised code C is allowed to call the platform's decrypting function on D_j, i.e., we must prevent the scenario of a third party stealing the agent's encrypted data and using the cryptographic service to recover m_j. To this end, m_j will always include an integrity token, A, computed from C. In other words, m_j will be a pair of the form (d_j, A), where d_j is arbitrary data. The platform's decrypting function will then check the presence and validity of the agent's integrity token before returning the decrypted data to its caller. Figure 1 shows the pseudocode for this decryption algorithm.

```
data decrypt( encrypted_data ) {

    contents = decrypt( encrypted_data, private_key );

    if (data_contents is a pair (d,A) AND
        integrity can be successfuly verified with A) {
      return d
    }
    else {
      destroy d
      error
    }
}
```

Figure 1. Platform decrypting function

4.2. Agent authentication

As shown in section 4.1, the platform decryption service relies on an agent integrity verification mechanism based upon an integrity token contained in the encrypted data. The easiest way of providing this integrity token A is to construct it as a cryptographic hash of the agent's explicit code C:

$$A \equiv H(C) \tag{1}$$

whith H a properly chosen digest function (such as SHA-1). Under this scheme, we can represent our agent as a pair (C, D), where D is split into platform-specific data chunks

$$D = D_1, \ldots, D_n \tag{2}$$

and, for each such chunk

$$D_j = E_{k_j}(d_j, H(C)), \tag{3}$$

where we follow the notation of section 4.1.

As a first result, it should be noted that this mechanism fully prevents any attack based on tampering with the mobile agent code C in charge of performing the decryption operations. Figure 2 depicts such an attack, where the agent's code C is modified so that it sends protected data to an unauthorised third party after it has been decrypted using the platform's cryptographic service. Such a modification would, however, entail a mistmach between the hash computed by the platform and that contained in D_j, and the subsequent abortion of the decryption process (cf. figure 1).

4.3. Data protection

The mechanism described so far protects transported data from agent's code (malicious) modifications. There exists, however, a second kind of security threat: injection of malicious data. We have seen how to protect data from code, but it remains to show how to protect code from data.

Our architecture allows, in principle, to wrap arbitrary executable code within the itinerant agent data payload D. In this situation, an external attacker could inject malicious code by replacing one of the agent's data chunks: it just needs to use both the right hash code ($H(C)$) and the plaform's public key to produce a D_j chunk with the expected format, $E_{k_j}(d', H(C))$, and without any restriction on its data contents d'. If C implements a generic algorithm decrypting D_j and blindly executing the code contained in d', the attacker will be effectively able to execute arbitrary code in the platform. In this scenario, the attacker leaves intact C, and is therefore able to forge an apparently valid data chunk D_j.

The key to prevent this kind of attack is to enhance D with additional structure, so that its validity can be checked by C. On behalf of interoperability and reuse, we need an algorithm that does not depend on the internal structure of the wrapped up code, allowing in this way an easy adapation of legacy agents to our general architecture. Let us describe such a general solution.

To begin with, the agent's developer generates an key pair (P_a, S_a) and embeds the public key P_a in the agent's explicit code C (e.g. as a static data member of an appropriate class). Each data chunk d_j is then signed using the private key S_a, and the resulting signature s_j is encrypted together with d_i and $H(C)$ to yield D_j; that is, we replace equation (3) by

$$D_j = E_{k_j}((d_j, s_j), H(C)), \tag{4}$$

where

$$s_j = E_{S_a}(H(d_j)). \tag{5}$$

In addition, the code in C that is in charge of data and code extraction follows these three steps:

1. Use the platform public decrypting function to extract (d_j, s_j) from D_j. This operation will suceed as long as C has not been modified after the agent creation.
2. Before using d_j, verify s_j against it, using the public key P_a (which is stored within the agent explicit code C).
3. If the above verification succeeds, use d_j. If this data wraps agent code, it will be executed at this point. If the verification fails, agent execution will be aborted and some alternative error handling routine will be called.

It is easy to show that this mechanism prevents malicious data injection inside the agent. Suppose that some malicious entity captures the intinerant agent while it is migrating between platforms and injects some bogus data within D, as described above. When the agent arrives at the platform, the architectural code C will extract the injected data, withouth any complaint on the platform's side. But, when C attempts to use the decrypted data, it detects that this data has not been signed by the agent's owner and aborts execution of the wrapped up code.

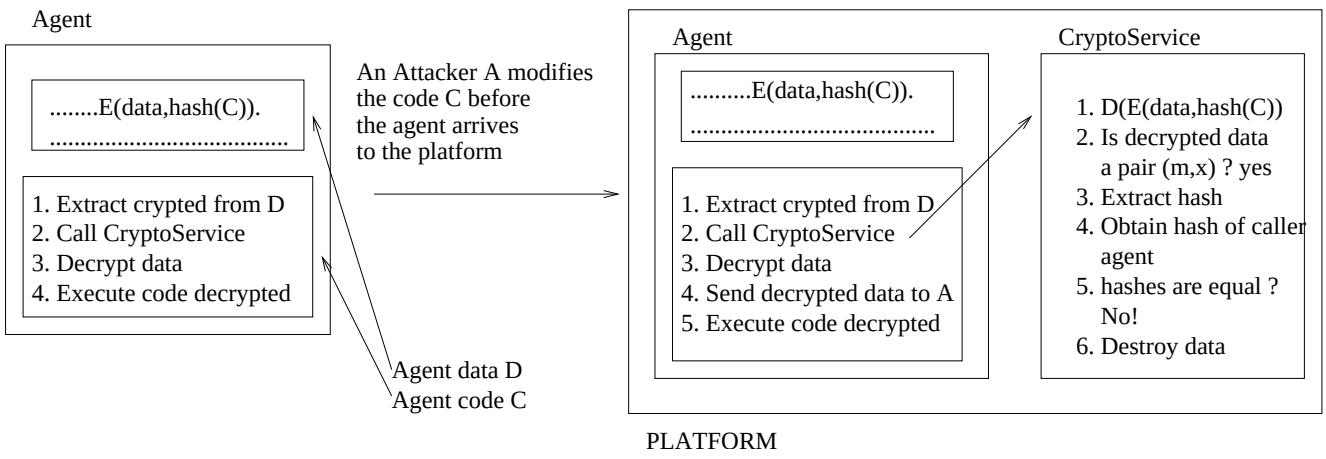

Figure 2. The encrypted hash of the code prevents code modification attacks

A more sophisticated attack would be injecting data inside the agent and simultaneously altering C, modifying the security checking code or the embedded public key P_a. This modified code could skip the signature verification of the decrypted data, and directly execute the (presumably malicious) wrapped up code. But we have already seen that this kind of attack is also doomed to fail: the agent will not be able to decrypt the tampered data, since its modification will be detected by the platform's decrypting function (due to the fact that C has been modified.)

5. Implementation

We have implemented the protection mechanisms described in this paper as a proof-of-concept add-on to *JADE*, a Java$^{\mathrm{TM}}$-based, FIPA compliant agent platform [1]. Each JADE agent platform can be split into distributed agent containers, and inter-container mobility mechanisms are already in place. Our first prototype extends these mechanisms to incorporate the cryptographic services described so far, and is part of an on-going effort to provide inter-platform mobility to JADE.

JADE agents are constructed by composition of so-called behaviours, which are instances of an abstract interface (`Behaviour`) that defines the agent services. Behaviour instances are an obvious match to our wrapped code data chunks, d_j. Thus, our add-on implements an adapter agent (`ItinerantAgent`) that contains the bytecode of behaviour implementation instances encapsulated according to equation (4). This adapter can be readily used to wrap new or legacy code implementing specific agent behaviours.

The unobtrusiveness of the framework is further enhanced via a factory class, `ItinerantAgentFactory`. It is worth noting that this factory differs from the well-known Factory design pattern in that its products are Java *classes*, rather than class instances. It takes advantage of Java's low level bytecode generation primitives to create on the fly new classes extending the `ItinerantAgent` with the public key P_a (section 4.3) embedded as a static data member. That is, given a (possibly already existent) set of behaviours, `ItinerantAgentFactory` creates the required class adapting them to the code protection mechanism presented in this paper. Therefore, the agent factory handles the creation of both the C and D components of the itinerant agent, making the process totally transparent to JADE application developers.

The platform's cryptographic service is implemented as an independent JADE agent (dubbed `CryptoAgent`), instantiated by each container. The `CryptoAgent` is accessed by itinerant agents via a FIPA-compliant ACL ontology, allowing the incorporation of our protection scheme with minimum fuss.

The implementation is completed with bare-bones PKI support to handle the creation and distribution of the required RSA keys and cryptographic mechanisms. Again, these services are implemented as independent, FIPA-compliant agents that can be registered in pre-exisiting JADE platforms without any kind of modification.

Summing up, we believe that our current implementation provides a concrete realization of our interoperability and reusability claims and strongly backs the viability of the proposed protection mechanisms.

Currently, further implementation efforts are directed towards a production-grade version of this inter-container mobility schema, and its upgrade to inter-platform migration scenarios.

6. Conclusions

Mobile agent protection can be achieved using cryptographic methods. The requisite verification and decryption tasks can be in charge of either the agent's code or the host platform. It has been shown that both platform- and agent-driven security, present drawbacks and apparently contradictory requirements. This paper has described a new solution for the protection of mobile agents that is based on a decryption interface provided by the platform, which is accessed by properly structured agents. It reconciles opposing requirements by introducing a hybrid software architecture that incorporates the advantages of agent driven proposals while limiting the impact of platform driven approaches. Interoperability, code reuse and deployment flexibility concerns are also fully addressed.

It should be stressed that even though this work could seem, at first sight, very specific, a deeper look reveals its potential to secure most of the currently deployed mobile agent applications with a very modest implementation effort. Existing solutions can adopt our architecture with a minimal development investment, and make the new protection mechanism coexist with pre-existing ones, if need arises. Deployment of secure platforms and, therefore, secure mobile agent applications is readily available, without prohibitive migration costs.

Another appeal of this view is that agents have the ability to manage their own code and internal structures. Apart from the obvious advantages of this situation regarding security, the same scheme can be applied to other common tasks, such as compression, codification, or interpretation. Moreover, the re-usability of code makes much easier the development of secure applications.

The whole scheme has been implemented and tested as an add-on to the well-known JADE platform, providing a down-to-earth realization of the proposed mechanisms. This implementation fosters our confidence in the viability of the protection scheme.

Future research on this field is envisaged through the definition of new architectures using all these mechanisms, and the deployment of new applications.

7. Acknowledgments

This work has been funded by the Spanish Ministry of Science and Technology (MCYT) though the project TIC2003-02041.

References

[1] Jade, java agent development framework. `http://jade.cselt.it`, 2004.

[2] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, 1996.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[4] F. Hohl. A Model of Attacks of Malicious Hosts Against Mobile Agents. In *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, pages 105–120, INRIA, France, 1998.

[5] W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000.

[6] Neeran M. Karnik and Anand R. Tripathi. Security in the Ajanta mobile agent system. *Software Practice and Experience*, 31(4):301–329, 2001.

[7] Chess David M. Security issues in mobile code systems. In *Mobile Agents and Security*, volume 1419, pages 1–14. Springer Verlag, 1998.

[8] J. Mir and J. Borrell. Protecting mobile agent itineraries. In *Mobile Agents for Telecommunication Applications (MATA)*, volume 2881 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, October 2003.

[9] S. Robles, J. Mir, and J. Borrell. Marism-a: An architecture for mobile agents with recursive itinerary and secure migration. In *2nd. IW on Security of Mobile Multiagent Systems*, Bologna, July 2002.

[10] V. Roth. Empowering mobile software agents. In *Proc. 6th IEEE Mobile Agents Conference*, volume 2535 of *Lecture Notes in Computer Science*, pages 47–63. Spinger Verlag, 2002.

[11] Volker Roth. On the robustness of some cryptographic protocols for mobile agent protection. *Lecture Notes in Computer Science*, 2240:1–??, 2001.

[12] B. Schneier. *Applied Cryptography*. John Wiley & Sons, New York, 1996.