Protecting mobile agents from external replay attacks

Carles Garrigues [a,*], Nikos Migas [b], William Buchanan [b], Sergi Robles [a], Joan Borrell [a]

[a] *Department of Information and Communications Engineering, Autonomous University of Barcelona, 08193 Bellaterra, Spain*

[b] *School of Computing, Napier University, Edinburgh EH10 5DT, United Kingdom*

Abstract

This paper presents a protocol for the protection of mobile agents against external replay attacks. This kind of attacks are performed by malicious platforms when dispatching an agent multiple times to a remote host, thus making it reexecute part of its itinerary. Current proposals aiming to address this problem are based on storing agent identifiers, or trip markers, inside agent platforms, so that future reexecutions can be detected and prevented. The problem of these solutions is that they do not allow the agent to perform legal migrations to the same platform several times. The aim of this paper is to address these issues by presenting a novel solution based on authorisation entities, which allow the agent to be reexecuted on the same platform a number of times determined at runtime. The proposed protocol is secure under the assumption that authorisation entities are trusted.

Key words: Mobile agents; Security; Malicious hosts; Replay attacks; Trip marker; Protected itinerary

1 Introduction

Mobile agents can provide many benefits to the development of distributed applications, but their use also poses many security threats. Research on mobile agent

* Corresponding author. Tel.: +34-935813577; fax: +34-935814477
Email addresses: carles@deic.uab.es (Carles Garrigues), n.migas@napier.ac.uk (Nikos Migas), w.buchanan@napier.ac.uk (William Buchanan), sergi.robles@uab.es (Sergi Robles), joan.borrell@uab.es (Joan Borrell).

technology has identified and solved a number of security-related issues, but there are still many remaining unsolved (Zachary, 2003).

Most of the research work undertaken on mobile agent security is concentrated on the problem of malicious platforms, as platforms have complete control over the agent execution, and can thus do almost anything with the agent code or data. Therefore, achieving a complete solution is considered impossible, although several problems can be mitigated. For example, even though platforms cannot be prevented from manipulating the results generated by the current execution of the agent, they can certainly be prevented from tampering with the results generated by the agent on other platforms.

Most proposed methods on mobile agent protection against malicious platforms try to provide a generic solution to cover as many security threats as possible, usually without putting the solution into practise in any real-life applications. On the other hand, most real-life agent-based applications do not take security into account. Subsequently, various proposals on mobile agent security have failed to address specific scenarios where their solutions may not be valid. In the context of this paper, the solutions presented in the literature for agent replay attacks have failed to address scenarios where the agent has to loop over a certain number of platforms an undetermined number of times (Tsipenyuk, 2004). Agent replay attacks can be classified in two different categories (Yee, 2003):

Internal replay attacks These occur when the agent is forced to execute repeatedly on a single platform using different inputs, aiming to obtain different responses and draw conclusions about its behaviour.

External replay attacks These are performed by malicious platforms by resending the agent to another platform, thus making the agent reexecute part of its itinerary.

Internal replay attacks are impossible to avoid because the platform has complete control over the execution, and can always reset the agent to its arrival state. On the contrary, external replay attacks can be avoided if platforms keep a record of previously executed agents.

The problem of current solutions (Yee, 2003; Westhoff et al., 1999; Wilhelm et al., 1998; Li et al., 2000; Suen, 2003; Mir and Borrell, 2003; Che et al., 2006; Cucurull et al., 2005) against external replay attacks is that they do not allow an agent to be executed n times on the same platform, especially if n is determined at runtime. However, the agent's itinerary often contains roundtrips that require the same platform to be visited several times. Thus, current solutions force programmers to sacrifice some of the inherent flexibility in mobile agent itineraries.

In order to enhance security and flexibility, this paper presents a solution based on *authorisation entities*. The advantage is that these entities are entitled to generate new identifiers for the agent, thus allowing the repeatable migration of the agent

to the same platform. Therefore, the proposed solution allows programmers to develop secure mobile agent applications and still maintaining the agents' intrinsic flexibility.

The rest of the paper is structured as follows. Section 2 introduces the related work on preventing replay attacks. Section 3 presents the proposed protocol for replay attack prevention. Section 4 describes the implementation of the proof-of-concept of the proposed solution. Section 5 concludes the paper and points out future directions.

2 Background

Replay attacks (Syverson, 1994) have traditionally been considered as a form of network attack. They are based on capturing some of the messages exchanged between two entities and sending them again at a later time. These attacks are usually performed during authorisation or key agreement protocols, in order to perform, for example, masquerade attacks.

Traditional mechanisms to prevent replay attacks are based on using nonces, timestamps, session tokens, or any other information that allows entities to bind their messages to the current protocol run (Aura, 1997). For example, an HTTP exchange between a browser and a server may include a session token that uniquely identifies the current interaction session. The token is usually sent as an HTTP cookie, and is calculated as a hash of the session data, user preferences, and so on.

Mobile agent systems are also exposed to traditional replay attacks. Any communication between two agents, or two platforms, or an agent and a platform is exposed to this kind of attacks. Mechanisms like the ones previously mentioned (based on nonces and session tokens) can be used to withstand these attacks.

In addition to traditional replay attacks, mobile agent systems are also exposed to *agent* replay attacks. Agent replay attacks are not based on replaying a message sent across the network, but on reexecuting an agent that has already been executed on a platform. In addition, these attacks are usually performed by platforms which are part of the agent's itinerary, and they are harder to prevent in this case. When agent replay attacks are performed by external platforms, they can easily be detected using authentication mechanisms (Navarro-Arribas and Borrell, 2006) or mechanisms designed to prevent traditional replay attacks (Karnik and Tripathi, 2001). Agent replay attacks can be divided into two classes: internal and external replay attacks.

Internal replay attacks occur when a dishonest platform repeatedly runs an agent with the same or different set of inputs each time. A platform might execute an

agent multiple times in order to understand its behaviour, or until the desired output is obtained. This type of attack is also known as *blackbox testing* (Hohl, 1998), and is usually performed when the agent code has been protected using some obfuscation technique.

This kind of attacks is performed inside a single platform, and cannot be externally observed by any other entity. Even if the agent tried to record all its actions on an external monitoring service, the execution environment could still interfere with these external communications, and route the messages to an incorrect recipient, or alter the contents of the messages, and so on. Moreover, agent attempts to store their state information in a secure external entity could be easily bypassed by malicious platforms altering the agent execution. In practise, internal replay attacks are impossible to prevent or detect (Yee, 2003).

External agent replay attacks occur when a dishonest platform propagates an agent to a remote host, without this migration being defined in the agent's itinerary. This kind of attack is especially difficult to deal with, as it is difficult to distinguish between a legal migration of the agent to its next destination and a replayed migration that the agent did not intend to do. For example, supposing that the agent's itinerary includes a migration from platform *A* to *B*, then platform *A* is authorised to send the agent to platform *B*, and the agent is also authorised to be executed on platform *B*. As a result, no authentication mechanism can be used to prevent platform *A* from maliciously resending the agent to platform *B* multiple times. This kind of attack can be carried out against a shopping agent, for example, in order to generate unintended purchases.

In order to provide a solution to this problem, Yee (2003) suggests considering a replay attack as an illegal state transition. Every platform within the itinerary implements a state transition inconsistency detection (STID) algorithm, which is able to determine when a migration from one platform to another is an illegal transition. The problem of this approach is that platforms must be aware of all possible illegal state transitions for every agent execution. In addition, illegal state transitions may be mistakenly identified if the agent is performing a loop in which the same platform is visited repeatedly.

In Vigna (1998), a protocol is presented that allows replay attacks to be detected upon completion of an agent execution. The protocol is based on recording the agent's execution on each platform. Platforms must keep a log of the operations performed by every agent, so that agent owners can detect any malicious manipulation of the agents' itinerary. This technique suffers from some drawbacks, such as the size of the logs that have to be maintained. In general, detecting replay attacks after they have been performed is useless in many occasions. For example, if the agent is buying a product, detecting a replay attack that leads to buying the product more than once is usually inappropriate, especially if the price of the product is too low to justify future legal actions.

Other work (Westhoff et al., 1999; Wilhelm et al., 1998; Li et al., 2000; Suen, 2003; Mir and Borrell, 2003; Che et al., 2006) on mobile agent protection against malicious platforms suggests the use of trip markers for preventing replay attacks. A trip marker is an agent identifier that must be stored by platforms, which can be used to detect and prevent future attempts of reexecuting the same agent. Again, the problem of these solutions is that they do not take into account the case where the agent itinerary includes one or more platforms that must be visited more than once. As a result, a legal reexecution of the agent in the same platform can be misinterpreted as a replay attack.

These issues were identified by Cucurull et al. (2005), who proposed a solution based on including counters inside the agent trip marker. Every platform is assigned a different counter, which indicates the maximum number of times that an agent can be executed on a platform. Platforms keep a record of which agents have been executed, and the number of times they have done so. Before starting the execution of an agent, platforms check that the number of times that the agent has been previously executed does not exceed the number of allowed executions stored in the agent trip marker.

The problem of this approach, however, is that the number of times that a given platform can be visited must be known in advance, specifically when the agent's itinerary is created, so that this information can be introduced inside the agent trip marker. Consequently, this approach does not allow the agent to dynamically decide on the number of times a given platform will be visited.

Preventing the agent from being executed more than once has also been referred in the literature as ensuring the *exactly-once execution property* (Straßer et al., 1998). This property is usually considered when designing fault-tolerant mechanisms for mobile agents. Ensuring the exactly-once execution property implies that, when the agent is launched to do a certain task: first, the task will be eventually executed, regardless of host or communication failures that may occur; and second, the task will not be performed more than once.

Solutions presented to ensure the exactly-once execution property are based on using external entities that monitor the execution of the agent. When a failure prevents the agent from continuing its itinerary, another agent is launched to resume the execution at the point the original agent left it. The problem of these solutions is that the communications between the agent and the monitoring system lead to considerable traffic overheads. In addition, these protocols severely reduce the agent's autonomy, as the agent has to constantly interact with the monitoring entity. Thus, they sacrifice one of the major advantages associated with the use of mobile agent technology.

To summarise, no solution presented so far against replay attacks allows an agent to dynamically determine the number of repeated executions of the same task on

one or more platforms along its itinerary. Considering that one of the greatest appeals of mobile agents is their dynamism and flexibility, hard-coding the number of possible migrations to a platform in advance can be a serious impediment for implementing real-life applications. The next section describes the proposed protocol which solves this problem.

3 Replay attack protection

Protecting mobile agents against all kinds of replay attacks becomes a serious and complex problem to solve. The simplest case is where the itinerary does not contain any loops in its itinerary, that is, the agent does not need to repeatedly execute the same task on one or more platforms. In a loop-free scenario, if the agent has to migrate to a certain platform several times, it will do so in different stages of its itinerary, which means that it will be executing different tasks. Replay attacks are easy to prevent in this case. Platforms can store an identifier of the stage or task executed, together with the agent trip marker. Fig. 1 shows an example of this kind of itinerary. In this case, *platform B* is visited in two different nodes (or stages) of the itinerary: the first one to execute node 2, and the second one to execute node 4.

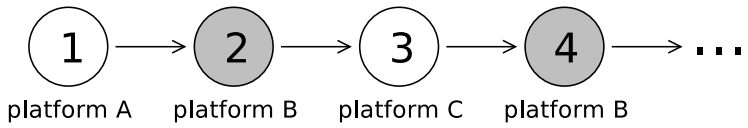

Fig. 1. Simple itinerary where the same platform is visited twice

It is worth noting that the usage of the term *node* in the context of this paper is different from that found in computer networks terminology, where a node simply refers to a location in the network. In this paper, an itinerary node, or, simply, node, refers to the stage of the agent execution that is associated with a certain task and a certain platform.

The most difficult case is where the agent has to execute the same task on the same platform several times, especially if this number of times is determined at runtime. Fig. 2 shows an example of this kind of itinerary.

As shown in the figure, the itinerary contains a loop, which starts at *platform B.* The detection of replay attacks is especially difficult, for example, when *platform C* resends the agent to *platform D*. How can *platform D* determine whether the new reexecution is a replay attack or a new iteration of the loop?

Before presenting the proposal for external replay attack prevention, the next subsection describes the requirements that any valid solution should fulfil.

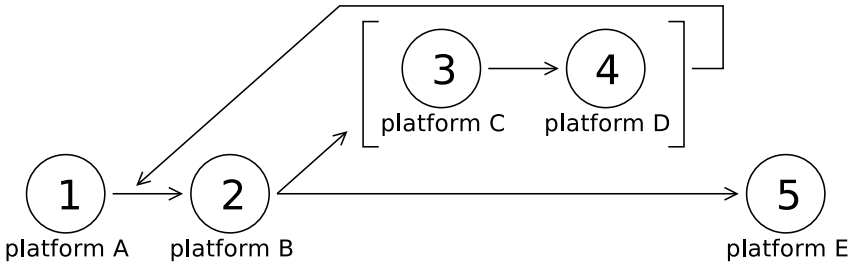

Fig. 2. Itinerary containing a loop with 3 platforms that are visited repeatedly

3.1 Requirements of the solution

First of all, different agents must have different identifiers or trip markers, even if they perform exactly the same tasks. This also implies that any new instance of the same agent must carry a different trip marker.

Secondly, a valid solution on replay attacks has to focus on their prevention, rather than just on their after-the-fact detection.

Thirdly, platforms must store the trip markers of the agents previously executed, along with an identifier of the task performed by the agent, to allow agents to revisit the same platform in different stages of its itinerary.

Finally, a valid solution on replay attacks should not involve the interaction of the agent with external entities. This allows the agent to run autonomously, without depending on the control or interaction with any monitoring service.

3.2 The protocol

The proposed protocol for the protection against replay attacks is based on using trip markers and authorisation entities. A trip marker is an *authorisation* that allows the agent to execute a certain set of nodes. Each node has an authorisation entity associated with it, and this entity is the only one entitled to generate new valid trip markers (authorisations) for the execution of that node. Platforms must store the trip markers of previously executed agents, so that no trip marker can be used more than once to execute the same node.

The steps required to create a replay-safe mobile agent, and then the operations carried out by the agent platform to detect and prevent replay attacks are presented in this section. The combined actions comprise the protection protocol which tackles replay attacks in mobile agent environments.

In order to create a replay-safe mobile agent, the programmer must define the set of nodes that comprise the agent itinerary, assigning the following information to

each one of them:

- a node identifier
- a task
- a node type
- an authorisation entity
- an authorisation node
- a set of next destinations

Although nodes can be assigned various types, for the purposes of the proposed protocol only two types are considered:

Regular A *regular* node is an itinerary stage where the agent executes its task and jumps to the next platform of the itinerary.

Loop A *loop* node is an itinerary stage where the agent decides whether or not to start a new iteration and revisit a certain number of nodes.

As an example, in the itinerary of Fig. 2, node 2 is a *loop* node, and the remaining nodes are all *regular*. However, as proposed in Straßer et al. (1998), other nodes types could be defined, for example, to provide more flexibility in the definition of the agent itinerary. These other types would be treated in the same way as *regular* nodes and, therefore, are not important for the definition of the proposed protection protocol.

In addition to a type, the programmer must assign an authorisation entity and an authorisation node to every node. The authorisation node is the node where the agent trip marker must be generated. This implies that the agent trip marker is only valid if it has been generated in the appropriate authorisation node. The authorisation entity is the corresponding platform or individual that must generate and sign the trip marker. In some cases, as explained next, a node can have no authorisation node associated with it, because the execution of that node is authorised by the agent owner.

The authorisation node that is assigned to each node depends on whether or not the node is located inside a loop. If the node is part of a loop, its authorisation node is the *loop* node located at the start of the loop. The corresponding authorisation entity is the platform where the *loop* node is executed. In all other cases, the node has no associated authorisation node, and the corresponding authorisation entity is the agent's owner. As an example, Fig. 3 shows which authorisation entity and node are associated with each node of a complex itinerary. This itinerary contains two loops, one nested inside the other.

The itinerary shown in Fig. 3 has an outer loop starting at *platform A*. In this platform, the agent decides how many iterations of the outer loop have to be performed. If the agent decides to enter this loop, it will visit *platform B* and then *platform C*, which is the starting point of the inner loop. The inner loop contains just one node,

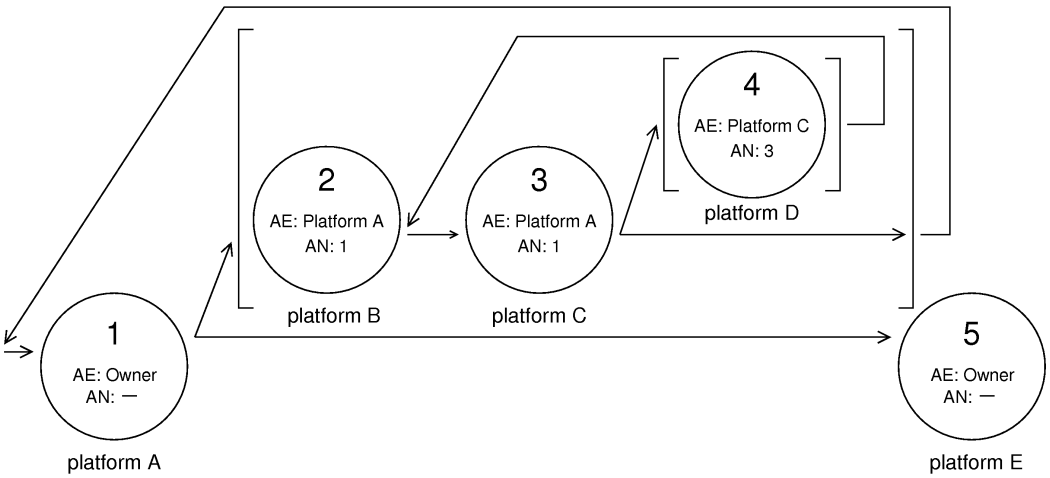

Fig. 3. Authorisation entities (AE) and authorisation nodes (AN) assigned to the nodes of a complex itinerary

executed on *platform D*. When the agent exits the inner and the outer loop, it migrates to *platform E*, where it reaches the end of its itinerary.

The itinerary shown in Fig. 3 has three different authorisation entities: the agent's owner, *platform A* and *platform C*. *Platform A* is the starting point of the outer loop, and thus it is the authorisation entity of nodes 2 and 3. On the other hand, *platform C* is the authorisation entity of node 4, as this node belongs to the inner loop. For the remaining nodes, the authorisation entity is the agent's owner.

It is important to note that the authorisation entities are selected by the agent programmer, and the proposed protocol assumes that the programmer trusts these entities to execute the *loop* nodes included in the itinerary. In the example of Fig. 3, the protocol assumes that the programmer trusts *platform A* and *platform C* to execute nodes 1 and 3 correctly. As mentioned in section 2, no protection mechanism can be used to prevent a malicious platform from subverting the agent execution, so the number of loop iterations could be easily altered as well.

Once the aforementioned information has been assigned to every node, the programmer must secure the itinerary using a protection protocol. This protocol must satisfy the following properties:

- It must not allow platforms to access or modify any part of the itinerary which is intended for other platforms.
- It must not be possible to introduce new nodes in the itinerary.
- It must not be possible to traverse the itinerary nodes in an order different from the order initially defined.
- Every itinerary node must be uniquely bound to the agent it belongs to. As a result, it must not be possible to reuse any part of the agent itinerary in a different agent.
- It must support flexible itineraries, which allow the agent to make decisions about

its travel plan at runtime.

In order to define the proposed protocol, it will be assumed that a unique agent identifier is used to bind the itinerary nodes with the agent. This agent identifier can be simply a timestamp attached to a big random number, or any other information that uniquely identifies each agent instance.

An example of a protocol that guarantees all the properties mentioned above is the one presented in Mir and Borrell (2003). This protocol is based on constructing the protected itinerary as a chain of digital envelopes, in such a way that they can only be opened by the appropriate platform in the correct order.

After defining all itinerary nodes and securing them using an itinerary protection protocol, the programmer must generate a trip marker for the agent. Agent trip markers must always contain the following information:

Agent identifier This is the same identifier included in the protected itinerary, which ensures that any given agent instance can be uniquely identified.

Authorisation node This is the identifier of the node where the trip marker is generated.

Expiry date This is the date after which the trip marker can no longer be used. It allows platforms to remove expired trip markers from their tables, which is usually convenient but has no real effect on the protocol.

Loop counter This counter is incremented by one unit every time the agent has to start a new loop iteration.

In the trip marker created by the programmer initially, the authorisation node is not set, as the programmer has no associated itinerary node. This is consistent with what has been specified in the itinerary nodes. When the authorisation entity of a node is the agent programmer, then the node has no associated authorisation node.

The trip marker initially created must be signed by the agent programmer or owner, who is always the authorisation entity assigned to the first itinerary node. This trip marker must then be placed at the top of the agent's trip marker stack. As will be seen later, the trip marker stack stores trip markers previously used by the agent during its execution. The top of this stack always contains the trip marker currently used.

Thus far, the steps required to create a replay-safe mobile agent have been described. Fig. 4 shows a representation of the components of an agent resulting from this protection. Next, the operations carried out by platforms to prevent a mobile agent from being replayed are presented.

In order to start the execution of an agent, platforms must extract the information of the current node from the protected itinerary. The operations carried out to do this extraction depend on the itinerary protection protocol used. In most cases, this

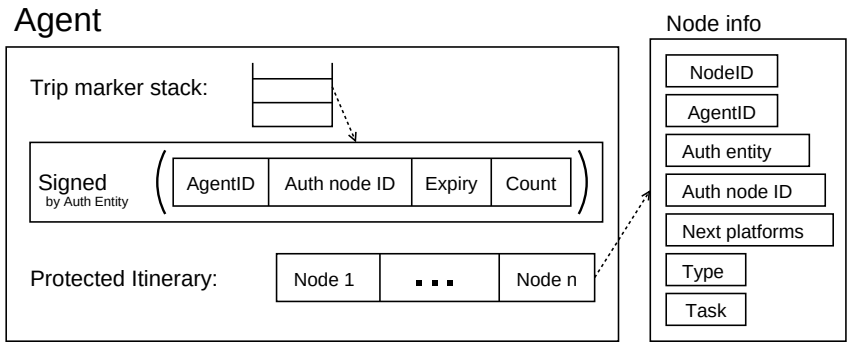

Fig. 4. Components of an agent protected against replay attacks

implies using the platform's private key, which ensures that no other platform can access the contents of the current node. By extracting the current node from the protected itinerary, platforms obtain the node identifier, agent identifier, task, type, next platforms, authorisation entity and authorisation node.

In addition, platforms must retrieve the agent's trip marker from the top of the agent's trip marker stack. The trip marker contains the agent identifier, authorisation node, expiry date and loop counter.

Platforms must also maintain a table with the trip markers of previously executed agents. Along with the trip marker, these tables must contain the identifier of the node executed by the agent. To effectively prevent replay attacks, platforms should only remove entries from their tables once they have expired.

In order to ensure that the agent is not being replayed, platforms must first use the agent's current node type to determine if the agent is executing a *regular* node or a *loop* node. In case that the agent is executing a *regular* node, platforms must perform the set of checks summarised in the algorithm of Fig. 5.

The most relevant lines of the algorithm of Fig. 5 are described in detail below:

Line 1 The platform uses a public key server to obtain the public key of the current authorisation entity. With this public key, the platform verifies the trip marker signature.

Line 2 The platform checks that the trip marker has been generated in the appropriate itinerary node. This involves checking that the authorisation node included in the trip marker is equal to that extracted from the current node.

Line 3 The platform checks that the agent identifier included in the trip marker is equal to the one obtained from the current node. This prevents a trip marker from being reused in a different agent, even if it was generated by the same owner.

Line 4 The platform checks the trip marker expiry date. If the trip marker has expired, the agent execution is discarded.

Line 5 The platform uses the agent identifier to look for a previous trip marker of the same agent in its trip marker table.

```
 1: verify trip marker (TM) signature
 2: check current auth. node = TM auth. node
 3: check agent's identifier
 4: check current date < expiry date
 5: check platform's TM table
 6: if agent was not executed before then
 7:    save TM in platform's table
 8: else
 9:    if node ident. is different then
10:       save TM in platform's table
11:    else
12:       if loop counter > previous one then
13:          replace previous TM from platform's table
14:       else
15:          discard agent execution
16:       end if
17:    end if
18: end if
19: run agent
```

Fig. 5. Algorithm for checking trip markers in *regular* nodes

Line 7 If there is no trip marker with the same agent identifier, the platform stores the new trip marker along with the current node identifier in its trip marker table.

Line 9 Otherwise, this means that the same agent was executed before on this platform. In this case, the platform checks if the agent's current node identifier is equal to that of the previous execution.

Line 10 If the new node identifier is different, it means that the agent is to execute a different itinerary node. The platform stores the trip marker along with the new node identifier in its trip marker table.

Line 12 Otherwise, this means that the same itinerary node was executed before on this platform. The platform then compares the current loop counter with the one included inside the trip marker obtained from the trip marker table.

Line 13 If the current loop counter is greater than the previous one, it means that the agent is performing a new iteration of a loop. In this case, the platform replaces the previous trip marker with the current one from its trip marker table.

Line 15 Otherwise, this means that the current trip marker has already been used, and the agent execution is discarded.

Line 19 If the agent execution has not been discarded in any of the previous steps of the algorithm, the current node's task is executed.

The above algorithm is necessary in order to verify that the agent is not being replayed when it is executing a *regular* node. When the agent is executing a *loop* node, basically the same checks are performed, but two additional operations are required.

```
1:  verify TM signature with the pubkey of either the
    current platform or the current authorisation entity
2:  check TM auth. node = current node or current
    auth. node.
3 to 18:  the same operations as algorithm of Fig. 5
19: generate new TM by incrementing loop counter
20: sign new TM with current platform's private key
21: if this is the first iteration then
22:     add new TM to agent's TM stack
23: else
24:     replace previous TM from agent's TM stack
25: end if
26: run agent
27: if agent exits loop then
28:     Remove TM from the top of stack
29: end if
```

Fig. 6. Algorithm for trip marker handling in *loop* nodes

Firstly, the platform must generate and sign a new trip marker for the agent. This new trip marker authorises the agent to execute all the nodes included inside the loop. The new trip marker is added at the top of the agent's trip marker stack, thus keeping the previous trip marker in the second position of this stack. This allows the agent to recover the trip marker used before entering the loop, which is essential because the trip marker generated by the current platform will no longer be valid when the agent exits the loop.

Secondly, if the signature verification performed in *line 1* of the previous algorithm fails, the platform will use its own public key to verify the current trip marker signature. This allows the agent to execute the current *loop* node using a trip marker generated by this same platform in a previous iteration.

As an example, in the itinerary of Fig. 3, *platform A* is the authorisation entity of nodes 2 and 3. For each iteration of these two nodes, *platform A* must generate and sign a new trip marker for the agent. In addition, it must verify the signature of the current trip marker with either its own public key or the owner's. Once the agent exits the loop, the trip marker signed by *platform A* must be removed from the top of the agent's trip marker stack, so that the original trip marker signed by the owner can be used again to continue the agent execution on *platform E*.

The algorithm shown in Fig. 6 summarises the operations carried out by the platform when the agent is executing a *loop* node. The most relevant lines of the algorithm of Fig. 6 are described in detail below:

Line 1 The platform verifies the agent trip marker signature with either its own public key or the public key of the current authorisation entity. If both verifica-

tions fail, the agent execution is discarded.

Line 2 The platform checks that the trip marker has been generated in the appropriate itinerary node. This involves checking that the authorisation node included in the trip marker is equal to either the current node or the current authorisation node. This allows the agent to use a trip marker generated by this same platform in the previous iteration.

Line 3 to 18 The platform checks the agent's identifier, expiry date, loop counter and node identifier, performing the operations 3 to 18 of the algorithm of Fig. 5.

Line 19 Before the agent task is executed, the platform generates a new trip marker. This new trip marker contains the same information as the previous one, except for the authorisation node, which is set to the current node, and the loop counter, which is incremented by one.

Line 20 The platform signs the resulting trip marker with its own private key.

Line 22 If the agent is to perform its first iteration of the loop, the new trip marker is added at the top of the agent's trip marker stack.

Line 24 Otherwise, the current trip marker, which was generated by this same platform in the previous iteration, is replaced by the newly generated one.

Line 26 The current task is executed, and the agent eventually decides whether or not a new iteration has to be performed.

Line 28 If no more iterations are required, the agent removes the trip marker from the top of the stack.

Thus far, the complete protocol aiming at preventing external replay attacks has been presented, describing the creation of a protected mobile agent and the operations performed by agent platforms to withstand replay attacks. In the next section, the most important characteristics of the proposed protocol are discussed.

3.3 Discussion

The proposed protocol is based on using trip markers (authorisations) which are generated at runtime by authorisation entities. The agent itinerary is comprised of a set of nodes, where each one is associated with a certain authorisation entity. Platforms only execute the current node of the agent itinerary if the trip marker has been signed by the appropriate authorisation entity.

Every itinerary node is also associated with an authorisation node. Using both an authorisation entity and node prevents the reuse of the agent trip marker in scenarios where the same authorisation entity is assigned to different nodes of the itinerary. In these cases, an attacker might try to replay an agent by reusing a trip marker signed by the proper authorisation entity but not generated in the appropriate itinerary stage. However, this would be detected by platforms when they check if the trip marker has been generated in the appropriate authorisation node.

Every itinerary node contains an agent identifier, which is also included in the agent trip marker. This prevents a given trip marker from being reused in different agents, as this would be detected by platforms when comparing the agent identifier extracted from the current node with the one included in the trip marker.

In order to prevent replay attacks effectively, the agent itinerary must be cryptographically protected, so that no attacker can change the information associated with a given node. Moreover, the itinerary protection must keep an agent task from being executed on a platform or itinerary stage different from that initially defined.

The proposed protocol does not involve the interaction of the agent with external entities, and only assumes that *loop* nodes will be executed on platforms trusted by the programmer. Because it is not possible to prevent platforms from tampering with the agent execution, it is legitimate to assume that the platform associated with a *loop* node is trusted by the programmer to evaluate the loop condition.

Platforms must prevent their trip marker tables from filling up by removing entries that have already expired. Additionally, some other policy has to be implemented in order to enable the removal of entries when no more trip markers can be added to the table (e.g., removing older entries first).

The proposed protocol supports free roaming agents that can decide their itinerary by themselves and can discover new hosts to be visited at runtime. However, the agent itinerary must be protected with a protocol that supports free roaming agents as well. An example of such protocol can be found in Garrigues et al. (2008). The proposed itinerary protection protocol is based on introducing trusted platforms into the agent's itinerary. The public keys of these trusted platforms are used to encrypt the agent's code and data to be executed on the hosts discovered at runtime. Then, when the agent is executed on a trusted platform, the itinerary is reconstructed using the public keys of the newly discovered platforms. The protocol described in this paper can be used in combination with the protocol proposed in Garrigues et al. (2008), provided that the platforms assigned to authorisation nodes are known at the time of creating the itinerary. Thus, the corresponding authorisation entities can be specified in the protected itinerary.

The proposed protocol does not support agent cloning. In general, mobile agent cloning introduces the problem of resolving identities properly when the replicas communicate with other agents or platforms. With regard to the proposed protocol, cloning a mobile agent would imply generating a new trip marker for the replica, however, this is not possible unless the new trip marker is generated by an authorisation entity. Nevertheless, authorisation entities are designed to generate new trip markers only when new loop iterations have to be started. As a result, the proposed protocol does not support agent cloning. Further research will be conducted in the future to extend the proposed protocol in order to allow for agent cloning.

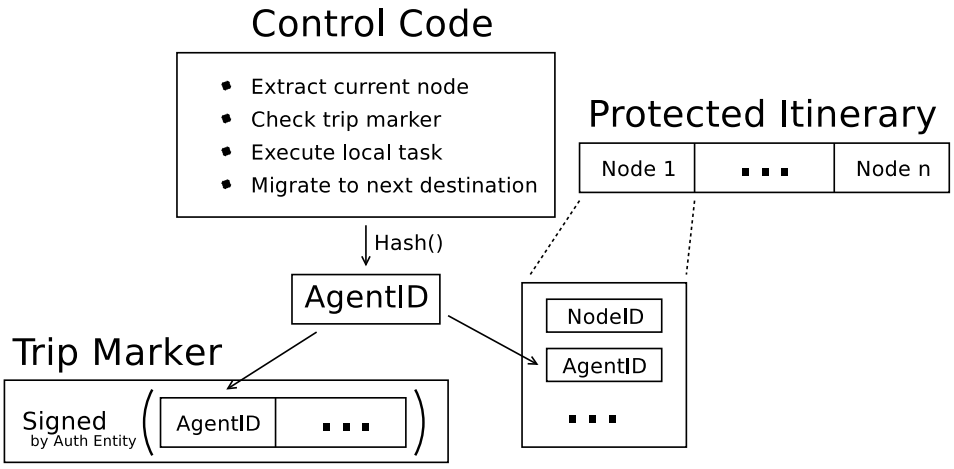

Fig. 7. Components of an agent protected against replay attacks with an agent-driven implementation

4 Implementation

In order to prove the viability of the proposed protocol, a prototype implementation has been created and experimentation has been carried out using the Java-based Jade platform (Bellifemine et al., 2003) as the agent execution environment.

Agents have been implemented following an *agent-driven* approach, as proposed by Ametller et al. (2004). According to Ametller et al. (2004), mobile agents are created from two main components: a protected itinerary and a control code. The protocol used to protect the itinerary is not imposed by Ametller's approach; the choice depends on the requirements of the given application. The control code manages the traversal of the itinerary, which involves the decryption of the protected itinerary data, the execution of the agent's tasks, and the agent's migration to the next destination specified in the itinerary. The advantage of using an agent-driven approach is that agents handle their protection mechanisms in an autonomous way, without requiring platforms to know how the agent is internally protected.

In this implementation, the itinerary is protected using the protocol presented by Mir and Borrell (2003). Fig. 7 shows the main components of an agent resulting from implementing the protocol presented in this paper using an agent-driven approach.

As this figure shows, the agent is made up by three components: the trip marker, the control code and the protected itinerary. These components are bound by an agent identifier, thus preventing from dishonest reuse in other agents.

As proposed by Ametller et al. (2004), the agent identifier is constructed as a hash of the agent control code. In addition, the control code is unique for every agent, and it is bound to the agent itinerary so that any malicious data insertion or modification

of the itinerary can be detected. At the same time, the itinerary data is encrypted in such a way that any change in the control code will keep this data from being decrypted.

In order to implement the proposed protection protocol based on an agent-driven approach, platforms must provide two services required by the agent control code. These services are available to the agent by exposing two functions:

decrypt This function allows agents to decrypt data using the platform's private key. As described in Ametller et al. (2004), data is bound to the agent using a hash of the control code. Thus, this function can check that the data being decrypted really belongs to the requesting agent.

checkTripMarker This function first checks that the trip marker really belongs to the requesting agent. For this purpose, it computes a hash of the agent control code and compares it with the agent identifier included in the trip marker. If this verification succeeds, this function validates the trip marker by performing the operations shown in the algorithms of Figs. 5 and 6.

As it can be seen, the trip marker check is triggered by the agent. Thus, the protection against replay attacks is completely optional, and will only be performed by those agents calling the *checkTripMarker* function. Forcing every agent to support the replay protection protocol would unnecessarily increase the complexity of many applications where security is not an issue.

In order to develop a better understanding of the construction of a replay-safe mobile agent, the following subsections describe each of the main steps required: creation of the control code, protection of the itinerary and generation of the agent trip marker.

4.1 Creating the agent control code

Both itinerary protection management and next platform decision making take place in the control code. The following are the relevant variables used by this code:

publicKey this variable is initialised to the Base64 encoding of a public key used for binding the control code with the itinerary, and making the control code unique at the same time (Ametller et al., 2004).

tripMarkerStack this variable will contain the agent trip marker stack, once the agent instance has been created.

protectedItinerary this variable will contain the agent protected itinerary, once the agent instance has been created.

The following are the relevant methods used by this code:

nodeExtraction this method extracts the information of the current node from the protected itinerary. The algorithm proposed by Mir and Borrell (2003) is used for this purpose. The platform's *decrypt* function is called to request the decryption of the itinerary data.

action this method is the entry point to the agent execution. It first calls the *nodeExtraction* method to extract the current node's information from the itinerary. Then, it calls the platform's *checkTripMarker* function to verify that the agent is not being replayed. Finally, it executes the current task and, eventually, triggers the migration to the next platform of the itinerary.

Once the agent control code has been created, it is compiled and an executable *.class* file is obtained. It is worth noting that the control code is almost the same for all agents, so it can be easily reused. However, the control code contains a public key which is different for each agent. Therefore, the hash of this control code is always unique, and it is a suitable agent identifier.

4.2 Creating the protected itinerary

After creating the agent control code, the protected itinerary is built. Every node of this itinerary contains: node identifier, agent identifier, authorisation entity and node, task, type and next platforms, as explained in section 3.2.

As proposed by Ametller et al. (2004), every node is now signed with the private key associated with the public key included in the agent control code. This private key is only available to the owner, and is different for each agent. Therefore, attackers cannot introduce new nodes in the itinerary nor reuse them in a different agent.

In order to meet the requirements specified by the replay protection protocol presented in this paper, the itinerary is now secured using protocol proposed by Mir and Borrell (2003).

4.3 Creating the agent trip marker and the final instance

After creating the control code and the protected itinerary, the initial agent trip marker is generated. This trip marker includes: agent identifier, authorisation node, expiry date and loop counter. All this information is signed with the owner's private key.

Once the control code, the protected itinerary, and the agent trip marker have been generated, a new instance of the agent can be finally created. The protected itinerary and the agent trip marker are supplied as parameters to this new agent instance, so

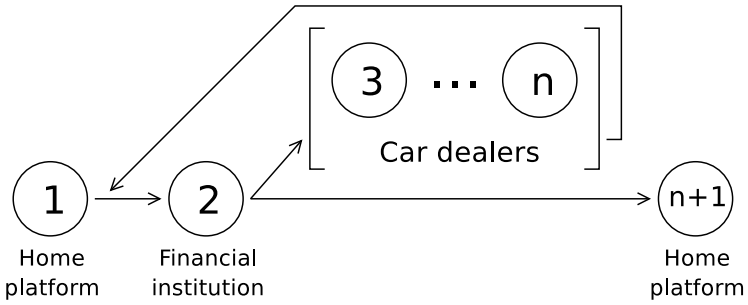

Fig. 8. Itinerary of the car purchasing service agent

that the *tripMarkerStack* and the *protectedItinerary* variables can be initialised.

4.4 Simulation and tests

In order to test the implementation of the proposed protocol, a simple mobile agent-based application has been simulated. This application shows the need for protection against replay attacks, and the results of the simulation prove that replay attacks can be effectively prevented, and agents can be executed in completely reasonable times.

This example application implements an automated car purchasing service. The service allows an individual to find the best price for a car, given its specific make and model, and the set of car dealers that have to be queried. The application carries out the purchase remotely from the user's financial institution, and this institution serves as the trusted location where the application decides which offer to accept.

Once the user has introduced all the required information in the application, a mobile agent is launched to visit every car dealer and obtain the price offered for the given model. After visiting all car dealers, a new round is started to negotiate an improvement on the best price previously obtained. This process is repeated until two consecutive iterations lead to the same best price. After each iteration, the agent visits the user's financial institution, to decide whether or not a new iteration has to be started.

When the best price is obtained, the agent proceeds with the car purchase. The purchase is always completed remotely from the user's financial institution, which ensures that this operation is performed securely in a trusted environment. Once the purchase has concluded, the agent returns to its home platform, and presents the resulting purchase contract to the user. The agent itinerary defined for the implementation of this example application is shown in Fig. 8.

This application demonstrates a simple scenario where the use of the mobile agent technology can introduce a number of advantages, such as reducing network load,

Table 1
Execution times (ms) for protected and unprotected agents

	Visited nodes:	15	27	39	51
Unprotected	Exec time:	11365	22094	32837	43665
agent	Time/node:	757	818	841	856
Protected	Exec time:	16641	31682	46715	61836
agent	Time/node:	1109	1173	1197	1212
	Increase:	46.4%	43.4%	42.3%	41.6%

by employing local communications, as well as automation of e-commerce processes. Considering that the negotiation can be a rather lengthy process, mobile agent technology enhances significantly the usability of this application in devices with intermittent, low bandwidth connections, as it eliminates the need for permanent connections with the remote sites.

This application also demonstrates the need of an agent to dynamically determine the number of repeatable visits required over a certain set of platforms. Unlike replay protection mechanisms previously presented, the proposed protocol allows the agent to visit the same platform as many times as necessary, without leaving it exposed to replay attacks. This is an essential part within the context of this application, as a malicious platform (e.g. an ill-intentioned car dealer) could easily resend the agent to its next destination to trigger additional car purchases.

In order to validate the proposed protocol, this application was simulated by introducing several malicious platforms in the agent's itinerary. These platforms acted as dishonest car dealers trying to trigger more purchases every time they could provide the best offer for a car. As expected, none of these attacks succeeded, because the next platforms of the itinerary immediately detected that the trip marker of the replayed agent had already been used.

The experiments performed also compared the execution times of the replay-safe agent with the unprotected agent in order to determine the overhead of the proposed protection protocol. The agent created for the tests was given an itinerary with three car dealers. The number of iterations required to reach the best price was varied in order to identify the overhead in terms of the agent's execution time. The evaluation setup used to make the tests was made up by 6 computers with 2 GHz Intel Pentium(R) IV processors and 256 MB RAM memory each. These computers were connected in a laboratory to a 100 Mbps Ethernet LAN. Table 1 shows the resulting execution times.

According to table 1, the execution time of a replay-safe agent is 43% higher than the execution of an unprotected agent on average. This increase, however, depends

greatly on the specifics of the application's requirements. In the context of this scenario, the agent negotiations with the car dealers took only a relatively short time. As a result, the time spent handling protection mechanisms significantly impacted the overall agent's execution time. In contrast, applications with high processing requirements may require the agent to execute time-consuming tasks, and thus, in that context, the time spent handling protection mechanisms would be negligible.

The increase in the replay-safe agents' execution times is readily reasonable, taking into account the complexity of the proposed protocol, and the added complexity of the itinerary protection protocol (Mir and Borrell, 2003). The overhead introduced by the execution of these protocols (around 350ms in absolute terms) is completely acceptable for this application. However, this may vary from one application to another. To conclude, it can be said that the implementation of the proposed protocol is not only advisable for any real-world application where security is an issue, but also perfectly feasible.

5 Conclusions

This paper presented a protocol aiming to protect mobile agents against external replay attacks. Previous published work on this area was mainly based on storing a trip marker, or some other kind of agent identification, inside agent platforms. This identifier was fixed for the whole agent execution, and, as a result, it was not possible to define itineraries where the same platform was visited more than once.

The proposed protocol protects agents against replay attacks, and allows agents to traverse itineraries that contain loops. The nodes that are part of a loop can be traversed repeatedly, an undetermined number of times. This allows programmers to define itineraries which take full advantage of the inherent flexibility of the mobile agent paradigm.

In order to make this possible, the proposed protocol is based on associating every node with an authorisation entity and node. The agent execution is only allowed if the agent trip marker has been generated and signed in the appropriate authorisation node, and by the corresponding authorisation entity.

The agent itinerary is cryptographically protected, in such a way that the authorisation entity assigned to a given node cannot be changed. Additionally, the trip marker contains a unique agent identifier that is also included inside every itinerary node. Thus, the trip marker of an agent cannot be dishonestly reused in different agents.

In order to prove the validity of the proposed protocol, implementation and experimentation work has been carried out using the Jade agent platform, and was

based on an agent-driven protection approach. The advantage of this approach is that agents were provided with a code that managed their own itinerary and protection mechanisms. As a result, agents can become more autonomous, and platforms can easily support the execution of agents with different protection algorithms.

The application simulated an automated, agent-based, car purchasing service with price negotiation and car purchase features. In such a scenario, replay-safe mobile agents are of utmost importance, as non-protected agents can be easily forced by malicious platforms to execute the purchase function multiple times, resulting in considerable loss of money. The simulation results prove that replay attacks can be effectively prevented, and replay-safe mobile agents can be executed in reasonable times. The expected increase in execution time was acceptable for the simulated application, despite representing 350ms of the total execution time for every node. In other applications with higher processing requirements, this overhead will likely be negligible.

Further work will be carried out to support agent cloning, and to develop tools that simplify the development of replay-safe mobile agents, by automating the protection of the itinerary and the protection against replay attacks.

Acknowledgements

This work has been funded by the Spanish Ministry of Science and Technology (MCYT) through the project TIC2003-02041, and the Spanish Ministry of Education and Science (MEC) through the project TSI2006-03481.

References

Ametller, J., Robles, S., Ortega, J. A., 2004. Self-Protected Mobile Agents. In: AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. IEEE Computer Society, pp. 362–367.

Aura, T., 1997. Strategies against Replay Attacks. In: Proceedings of the Computer Security Foundations Workshop. IEEE Computer Society, pp. 59–68.

Bellifemine, F., Caire, G., Poggi, A., Rimassa, G., 2003. JADE - A White Paper. Tech. rep., Telecom Italia Lab, available at http://jade.tilab.com.

Che, H., Li, D., Sun, J., Yu, H., 2006. A Novel Solution of Mobile Agent Security: Task-Description-Based Mobile Agent. IJCSNS International Journal of Computer Science and Network Security 6 (2B), 121–125.

Cucurull, J., Ametller, J., Ortega-Ruiz, J. A., Robles, S., Borrell, J., 2005. Protecting Mobile Agent Loops. In: Mobility Aware Technologies and Applications. Vol. 3744 of Lecture Notes in Computer Science. Springer-Verlag, pp. 74–83.

Garrigues, C., Robles, S., Borrell, J., 2008. Securing dynamic itineraries for mobile agent applications. Journal of Network and Computer Applications doi:10.1016/j.jnca.2007.12.002.

Hohl, F., 1998. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In: Mobile Agents and Security. Vol. 1419 of Lecture Notes in Computer Science. Springer Verlag, p. 92.

Karnik, N. M., Tripathi, A. R., 2001. Security in the Ajanta mobile agent system. Software Practice and Experience 31 (4), 301–329.

Li, T., Seng, C. Y., Lam, K. Y., 2000. A Secure Route Structure for Information Gathering Agent. In: Proceedings of the 3rd Pacific Rim Int. Workshop on Multi-Agents: Design and Applications of Intelligent Agents. Vol. 1881 of Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 101–114.

Mir, J., Borrell, J., 2003. Protecting Mobile Agent Itineraries. In: Mobile Agents for Telecommunication Applications (MATA). Vol. 2881 of Lecture Notes in Computer Science. Springer Verlag, pp. 275–285.

Navarro-Arribas, G., Borrell, J., 2006. An XML Standards Based Authorization Framework for Mobile Agents. In: Secure Mobile Ad-hoc Networks and Sensors. Vol. 4075 of Lecture Notes in Computer Science. Springer Verlag, pp. 54–66.

Straßer, M., Rothermel, K., Maiöfer, C., 1998. Providing Reliable Agents for Electronic Commerce. In: Proceedings of the International IFIP/GI Working Conference. Vol. 1402 of Lecture Notes in Computer Science. Springer-Verlag, pp. 241–253.

Suen, A., 2003. Mobile Agent Protection With Data Encapsulation And Execution Tracing. Ph.D. thesis, The Florida State University.

Syverson, P., 1994. A Taxonomy of Replay Attacks. In: Proceedings of the Computer Security Foundations Workshop. IEEE Computer Society, pp. 131–136.

Tsipenyuk, Y. Y., 2004. Detecting External Agent Replay and State Modification Attacks. Master's thesis, University of California.

Vigna, G., 1998. Cryptographic Traces for Mobile Agents. In: Mobile Agents and Security. Vol. 1419 of Lecture Notes in Computer Science. Springer-Verlag, pp. 137–153.

Westhoff, D., Schneider, M., Unger, C., Kaderali, F., 1999. Methods for Protecting a Mobile Agent's Route. In: Proceedings of the 2nd Int. Information Security Workshop (ISW '99). Vol. 1729 of Lecture Notes in Computer Science. Springer-Verlag, pp. 57–71.

Wilhelm, U. G., Staamann, S., Buttyan, L., 1998. On the problem of trust in mobile agent systems. In: Proceedings of the Symposium on Network and Distributed System Security. Internet Society.

Yee, B., 2003. Monotonicity and partial results protection for mobile agents. In: Proceedings of the 23rd Int. Conf. on Distributed Computing Systems. IEEE Computer Society, pp. 582–591.

Zachary, J., 2003. Protecting Mobile Code in the Wild. Internet Computing, IEEE 7 (2), 78–82.