A Security Oriented Architectural Approach for Mobile Agent Systems

Stamatis Karnouskos
German National Research Center for Information Technology
Research Institute for Open Communication Systems (GMD-FOKUS)
Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
Email: karnouskos@fokus.gmd.de

ABSTRACT
Security is a critical parameter for the expansion and wide usage of agent technology. A threat model is constructed and subsequently the basic techniques to deal effectively with these threats are analyzed. Then this paper presents a dynamic, extensible, configurable and interoperable security architecture for mobile agent systems. It is explained how this architecture can be used to tackle a big par of security threats. All the components of the security architecture are analyzed while we also argue for the benefits they offer.

Keywords: Mobile Agents, Security Architecture, Security Threats

1. INTRODUCTION

Security means think negative! People dealing with security have a hands-on experience with such issues. We can't really expect that the systems designed and developed will be used according to the predicted/desired usage. On the contrary, you have to think of all cases (if that is possible) that something might go wrong. If there exists even the slightest possibility for a security breakout, then you can be sure that someone sometime will find and take advantage of it.

A secure system is a system that provides a number of services to a selected group of users and restricts the ways those services can be used. A security service is a software or hardware layer that exports a safe interface out of an unprotected and possibly dangerous primitive service. In order to build a security service we need a security architecture. Having analyzed the security needs of the Mobile Agent (MA) technology we propose in this paper a dynamic, extensible, configurable and interoperable security architecture for mobile agent systems.

Software agents [1] are a rapidly multi-directional developing area of research since the early 90s. Yet research community has not been able to find a clear answer to the most popular question "What exactly is an agent?" and the debate still goes on. A general answer could be: Agents are software components that act alone or in communities on behalf of an entity and are delegated to perform tasks under some constraints or action plans. Mobile Agents shatter the notion of client/server model and eliminate its limitations. Standardization efforts and guidelines that boost the usage of agent technology exist in organizations such as the Object Management Group [11] and the Foundation for Intelligent Physical Agents [12]. Agents are computer and transport independent (they depend only on the execution environment) and therefore promote interoperability among systems and software.

2. THREATS IN A DISTRIBUTED AGENT ENVIRONMENT

Mobile code programming is by its nature a security-critical activity. In an agent based infrastructure the security implications are far more complex than in current static environments. In such an environment author of the MA code, the user, the owner of the hardware, the owner of the execution platform (even the execution place) can be different entities governed by different security policies and possibly competitive interests. In such a heterogeneous environment security becomes an extremely sensitive issue.
We identify the threats that exist in an agent-based infrastructure. Later we will demonstrate how our design attempts to handle these threats. We can have : misuse of execution environment by mobile agents, misuse of agents by other agents, misuse of agents by the execution environment, misuse by the underlying network infrastructure.

Mainly all security efforts target the first category and a big part of the second one. But misuse of agents by the host isn't touched almost at all. Ongoing work on the subject [2][9] may provide help in effectively targeting this area also. Our approach also provides protection for the two first categories and tries to provide some guarantees to the agent concerning the host code and execution environments. Though this can't be an integrated solution as it tackles only a small part of the problem widely known as the "malicious host" problem.

2.1 Misuse of Hosts by Mobile Agents

Malicious agents while visiting a host can :

- Destroy/reconfigure/change or even erase resources of the host. This affects all agents visiting the host that time. With various tricks or false language implementations [17] an agent can bypass authorization and authentication stages and obtain access to private data.
- Cause denial of service attack. The agent overloads the host e.g. by consuming all network resources and then the host can't provide the expected services to the other agents.
- Eavesdrop. The agent can access sensitive information on the host e.g. the private key of the host, modify the security policy in order to obtain more access rights etc.
- Masquerade. The agent can pretend being someone else and therefore be objected to the wrong policy schemes.
- Violate non-repudiation. An agent can deny performing several actions to the node.
- Perform complex attacks. Here more than one agents co-operate in order to attack a host. These are the most difficult attacks as they are strategically planned and can be event triggered. This collaborative kind of attacks are very difficult to identify no to mention to deal with them.

2.2 Misuse of Agents by Other Agents

An agent can attack an other agent by changing agent's internal state, accessing/changing data of an agent (e.g. access the memory where agent keeps its own data), trapping an agent and changing its mission, stealing info, claim a false identity and in purpose damage agent's reputation, delay an agent in order to distract it from its goals etc

2.3 Misuse of Agents by Hosts

A host can have complete control over an agent. There for it can change his objectives, provide wrong execution and return wrong results, steal/change internal data e.g. electronic money or offers (if it is an auction agent), delete an agent or suspend it for enough time so that the operations the agent wanted to perform are not valid any more or have no meaning. E.g. an auction agent that has missed an auction can't really fulfill its goal

We mentioned above the main threats that exist in an agent-based infrastructure. Of course a combination of them makes it even more difficult to prevent or deal successfully with it. Also all above mentioned security breakouts are performed when an agent is visiting a host. Contrary to popular belief agents don't transport themselves to the next host. So in any case the agent relies on the agency to transport its code safe and secure to the desired host!

2.4 Misuse by the Underlying Network Infrastructure

Threats exist also while the agent traverses the network from host to host. One external attacker could perform all kind of attacks such as agent deletion/alteration/copy & replay/stealing etc. A not so superficial scenario is the following: Agencies are run by a user e.g. in a Unix host. By misconfiguration the user that runs the agency allows others to access and modify the files that are stored on disk e.g. the policy files. Then another user could easily change the policy file and allow his agents to execute. The difficulty with this kind of attacks is that can't all be dealt because they use other resources than the specific product does. A product that runs in a Unix environment is vulnerable to all kind of attacks via the security holes of the Unix system. Such kind of attacks can't be predicted by the designer of the agent platform and are also out of the scope of this paper.

3. DEALING WITH THE SECURITY RISKS

Having presented the threat model we will try here to see how we can deal with these problems. There are four main security requirements to be satisfied:

Confidentiality. Private data carried by the agent or used by the platform (such as audit logs) should remain private. Intra- and inter- platform communication should by no mean be revealed to 3rd parties by monitoring or other techniques.

Integrity. Agent code should be protected from unauthorized or accidental modification of code, state and data. If that is not possible it should be at least pragmatic to detect agent tampering. The platform should take the same countermeasures.

Accountability. Agents and platforms should audit their activities and be able to provide detailed info for debugging or security purposes. Every action should be uniquely identified, authenticated and audited.

Availability. Resource management, controlled concurrency, deadlock management, multi-access, detection and recovery from faulty states such as software and hardware failures apply to mostly to platforms. Agent should also be able to monitor their services and actions in order not to be driven to endless loops.

Several approaches have been developed in order to minimize security risks. We will not examine those approaches, instead we will focus on cryptography, signing and policy.

3.1 Cryptography

The basic purpose of cryptography and specifically encryption is to guard sensitive data against unauthorized access from non-intended recipients. Encryption techniques are used to acquire features such as :

- Data confidentiality and secrecy since all messages have to be decrypted in order to process the enclosed info.
- Data integrity, because if the cipher text has been tampered it won't be possible to decrypt correctly the original message
- Authentication. Taking for granted that the secret key of the signer remains secret, we can be sure that the one who signed the data is who he claims to be.
- Non-repudiation. Public key technology can provide non repudiation of the recipient and its actions.

One-way hash functions, symmetric and public key cryptography belong to encryption techniques. Encryption is used in order to strengthen security. The user should be able to chose from a wide variation of encryption algorithms and have the ability to implement his own and make it available to other users. The Component database that exists in our architecture ensures exactly that. Encryption guarantees authenticity, integrity, and secrecy of data and communication.

3.2 Credentials and Authentication

Because agents are programs, they are intangible and live in a virtual world, we connect the trust model of such an infrastructure with the trust model of real world in order to make security critical decisions. That basically means that since every agent acts on behalf of a user or generally an entity we check to see if we trust that entity and indirectly trust the agent. The connection between those two worlds, the virtual one of agents and the real one is done via the digital certificates. A digital certificate is an object (file or message) signed by a certification authority that certifies the value of a person's public key. X509 [3] certificates of the International Standard Organization are the most popular, so we also adopt them in our design.

An agent is signed by one or more entities. Those entities can be either the creator of the code, the user that dispatched the agent (usually this is also the creator), a place of a host and generally any entity that holds a valid certificate.

Signing an agent guarantees that i) the creator is the one claimed by the agent, ii) agent's code (at least the signed part) has not been tampered by a 3rd party during transportation. Signing doesn't guarantee that the agent will execute correctly (safety). Furthermore one place can encrypt the agent with the public key of the destination place (only the destination place has the private key to decrypt the agent), protecting in this way the agent while it traverses the net until it reaches the final destination.

In order to ensure secure external communication we don't use any homegrown solutions but instead we use the SSL (Secure Socket Layer) protocol [4]. TLS standard (Transport Layer Security) [5] is also another option.

Credentials also touch indirectly the "malicious host" problem. Since each place (or at least each agency) has its own certificate there is proof that this agent is mapped to a legal user who bears responsibility of the behavior of the agency. An agent (based on a trusted host) prior to transportation can get next host's credential and decide whether to migrate and what to compute on the specific host. Furthermore it can ask the place to sign the results with its private key, so it can prove that those results were obtained during the execution on that specific place (repudiation problem). Non-changing parts of the agent should be signed for maximum protection.

So we see that by depending on the certificates we can extend our dependency in the real world where each entity exists or at least has a person that is responsible for the actions. The disadvantage of using certificates is that it assumes an advanced stage of existence of public key infrastructure which has its own pros and cons [16].

3.3 Access Control Checks

Having successfully identified the agent is only the first step. Trust in the agent's credentials doesn't guarantee that it will

behave legitimate nor execute correctly. Thus we monitor and authorize every call it makes to platform's resources. Any access to any resource e.g. network, file, system configuration etc is subject to a access control check. Therefore we need a policy and an enforcement manager to make sure that our policy is enforced. With this second level of check we provide fine-grained control customized per user or group. As users perform various activities not all of them have the same rights. The security is based in protection domains of Java. Those protection domains are defined by the internal agent id (not immutable) and/or by the signer(s) of the agent code (immutable). We can even require a combination of user identities in order to allow an agent to perform a task. A flexible policy scheme guarantees exactly that. Although this second level provides some extra and selective security we understand its limits. Even though we restrict what the agent can do, we can't be sure that no harm will be caused on purpose (e.g. buffer overflow) or by mistake or wrong execution e.g. via random side-effects.

3.4 Code Verification and Java

We try to verify that the code of the agent arriving to our agency is valid. That means that the bytecodes refer to valid instructions. This is one of the reasons why we use Java. Java is a very popular language for implementing agents. The motto "write once and run anywhere" gains momentum. Generally we chose Java as the implementation language because of its features such as: language design with security in mind, byte code verifier, dynamic loading, strictly typed language, lack of pointer arithmetic, automatic memory management including garbage collection to avoid memory leaks and dangling pointers, check of array references to ensure that they are within the bounds of the array, strong typing etc. Furthermore Java is widely used and evolvable. That's a non-technical characteristic of the language we need. This is not for commercial/political reasons but for practical ones. A language used by a small group of people might be task-specific but it would be difficult to advance and keep up to date. Also bugs, errors, misbehavior would be seldom if at all reported. Thus we need a language that is widely used so that it evolves fast and day by day new features are added constantly depending on the needs. Also platform independence is not mandatory but would be of great help since our efforts could be ported/deployed easily and quickly to a heterogeneous environment.

Java features also a byte-code verifier in order to ensure that Java-written agents won't perform illegal instructions e.g. writing out of the memory space. On the other hand bytecodes are more expressive than Java. That means that there is valid bytecode representation for which Java code is not valid. With JASMIN [6], or similar tools freely available in internet, one can write illegal bytecodes that illegally mess up with Java's semantics. As we see Java isn't the panacea and there are security problems but as long as they are discovered and corrected in the next versions we will be sure that our system's security is also strengthened. We understand also that this language has its limitations and weaknesses. It is not the perfect language specifically designed for mobile agents but is good enough. Problems like the greater expressiveness of the bytecode verifier, implementation errors of the Java's native code, or even other bugs that pop up every day put the language in a continuous test via which errors are corrected and we can hope for a strengthened security as well as increasing performance in the future.

4. THE SECURITY ARCHITECTURE

Security can't be an afterthought! It has to be integrated with the Agency's core functions and not implemented at the end as an extra, optional, explicitly called service. Approaches that try to incorporate security after the design phase have been proven to fail. The security architecture (Figure 2) for mobile agent systems tries to incorporate all above solutions to the threat model presented before and also to be as open as possible in order to integrate easily future solutions. Furthermore we follow in this approach the MASIF standard for interoperability reasons.

4.1 Places

The agent system (Figure 1) consists of places. A place is a context within an agent system in which an agent is executed. This context can provide services/functions such as access to local resources etc. A place is associated with a location which consists of a place name and the address of the agent system within which the place resides. Places can contain other places. All places follow the parent-child paradigm of Unix processes in the way that each child is assigned/makes use of its parents resources. Also its policy is an extension/customization of its parent's policy.

A place can be used in different ways. Places are i) dynamically assigned to agents as they enter the agency based on some criteria e.g. all agents coming from a specific user or location or agents belonging to a specific policy scheme etc. or ii) statically (permanently) assigned per entity (e.g. user, enterprise etc). In the latter static resources are given to the place (after agreement with the node provider) and the local resource manager manages them. With this way it is possible for an enterprise to setup a network of places in various nodes, creating a Place-Oriented Virtual Private Network [13]. This offers several advantages e.g. secure communication or paths between company-trusted agents etc.

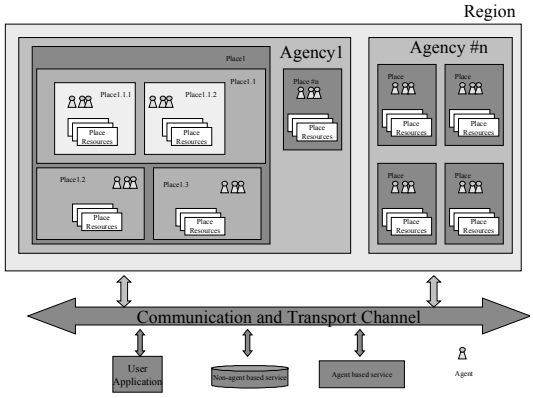

Figure 1 - The distributed agent environment

A policy scheme and a resource access scheme are assigned to each place and the respective policy and resource manager are given the general security guidelines, which can never be bypassed. If an agent has sufficient credentials, then it can fully interact with the components e.g. change the place's policy, ask for more resources, insert elements in the component database etc.

The existence of different Execution Environments (EEs) for agents that have the same owner/characteristics serves the need to avoid unwanted interactions. Isolation done by EEs is similar to the sandbox idea that exists in Java. Since in each place agents with common characteristics (e.g. of the same owner) are gathered the possibility of attacking each other is lower than usual. Of course advanced security facilities offered by the place can be used to minimize these risks (e.g. a secure communication service via the platform). Furthermore if one wants can use a place as a TestPlace (a firewall like approach) and allow suspicious agents to execute there, monitor the results and then determine if it will allow them to execute in the real place. Certainly if you see for instance that an agent changes inappropriately the policy file of the TestPlace you forbid it to execute into the desired place

(which otherwise would be catastrophic). Also agents are somehow isolated since each one has its own classloader.

Places beyond having unique IDs, also hold their own public/private keys. An agent can ask to be signed in order to have a proof that it passed via this place. This also helps with the so-called "multi-hop" security problem. If every place signs a specific part of the agent then we can trace back the exact route the agent followed. Based on that info we can take further security decisions. Let us mention that if there is one malicious host who tries to break the chain of valid signatures (not sign the part of the agent because he performed something maliciously and doesn't want to leave any traces) it will be detected by the next non-malicious place.

4.2 Policy Manager

The Policy Manager is responsible for managing the policy schemes stored in the policy database. By separating the policy DB from the enforcement engine we insert a dynamic way of policy modification. As our system's in progress implementation is based on Java, we use the policy language supplied by JAVA2 for interoperability reasons and extent it whenever we see need to do so (e.g. new type of access rights to use the credential DB etc). The security policy defines the access each piece of code has to resources. Signed code can run with different privileges based on the identity of the person or place who signed it. Thus users can tune their trade-off between security and functionality (of course within limits given by administrator).

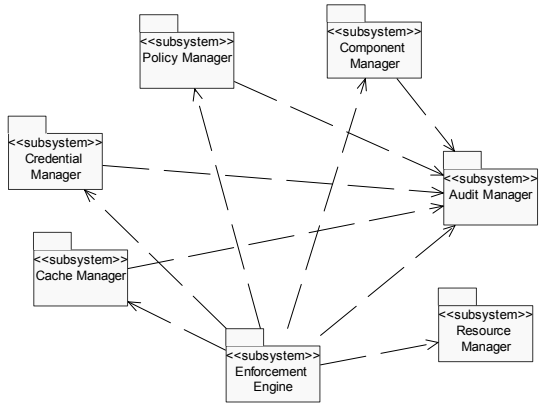

Figure 2 - UML [15] representation of the security architecture's basic components

When an agent comes to an agency then he is subjected first to the general agency's policy which is set by the user that initiated the agency (Figure 3) and is considered to be the super-user. Subsequently after passing successfully that control the agent is subjected to the place's specific policy. It is clear that with this sequential check of policies we avoid the problem of granting contradictory access rights for the same action by different policies. The policy of the father place is always first checked and therefore it has precedence over child's place policy. This architecture makes it easy for an enterprise to set-up an agency and then provide advanced services to its customers. One of those services is to provide places which are managed by the customer and don't violate the general rules of usage set by the enterprise. Having this way of thinking in mind, one can easily understand the hierarchical policy structure and its implications presented here .

Notification of malicious agents (that have attacked other hosts) can be distributed in the network (like CERN security notifications). When our agency receives such a notification it can add a line to agency's general policy (that is always checked first) that will not allow agents that bear those malicious characteristics to migrate to any of the hosted places. E.g. it will not migration to all agents signed by a user considered as malicious. One can also simply forbid agents from a specific user/domain for personal reasons e.g. because they consume too many resources, or belong to a competitor etc. This is a kind of local black list which in co-operation with the local certificate revocation list provides a higher level of flexibility and customization of the system.

Any attempts to describe the security policy in terms of each individual principal's authority to access each individual object is not scalable and not understandable for those instituting the policy. Thus it has been proposed to group principals and objects into sets with common attributes, where the attributes are used in making security decisions rather than the individual identities. So we have role-based policy, group policy, clearance labels, domains etc. Furthermore by grouping policies we allow for faster execution times while trying to enforce the policy. In our system all security checks are identity-based in order for an agent to enter a place. After an agent successfully enters a place future security checks become role-based. Thus we don't have each time to verify agent's credentials. We check only to see in which place the agent resides and what is the appropriate policy for that place. This approach is once more followed in our effort to speed up security checks and improve architecture's performance.

4.3 Credential Manager

Credentials are used to i) verify that the component was created/distributed by the claiming principals, ii) verify that the component hasn't been altered after it has been signed, iii) fulfill partially the non-repudiation need so that the originator of that code can't deny it.

Credentials are stored in the credential database. All actions concerning the credentials (including management of the credential database) are handled by the credential manager (CM) . The CM checks the validity of the certificates, updates them, maintains the local revocation list etc. The local revocation list acts as a second black list only that this time the user can locally make invalid the agent's certificates and therefore force the system to treat the agent as an anonymous one. While the first list forbids migration to the agency (via SSL authentication) here we have only sandboxing of the agent (treated as possibly malicious).

X509v3 Certificates [3] are used as credentials in a heterogeneous environment with a key used as the primary identification of a principal. Other certification systems beyond X.509 could be used e.g. PGP or SKIP but none of them could be considered as superior to the others as their features as well as design and usage logic vary greatly [8].

In our approach we assume that users have certificates and that hosts also have certificates. Places can also have certificates in order to sign results. As the nested-place approach we take is service oriented (place n can belong to a different provider than the sub-place n+x), we can ask from the n^{th} place to sign a part of an agent. If that place doesn't have a certificate, it can use (if permitted by policy) the certificate of place n-1 or if that place also doesn't have a certificate then that of n-2 etc. Finally if also the host doesn't have a certificate or somewhere between the policy of place k (with $1<k<n$) forbids the use of a certificate from parent places then the action fails.

The certificates of course assume the existence of a public key infrastructure with certification authorities (CAs) which issue certificates that bind two principals in a speaks-for relationship. When checking the validity of certificates the credential manager looks up firstly his local database and his local revocation list. In the local databases a copy of the previous certificates of user's agents that have executed exist. This is done for performance reasons. If the local lookup action returns with an error (meaning

that certificate doesn't exist locally) then via the use of a protocol e.g. LDAP/LDAPS [14] its validity is checked in cooperation with a CA server, and the results are stored in the local database in a time-limited manner for future reference.

4.4 Component Manager

The Component manager mainly manages all requests concerning components preinstalled by the administrator as well as user installed components in the component database. The component manager allows first the administrator to install code and selectively via policy make it available to the users. This code can be signed so that agents coming to the agency can verify the originator of the code and decide whether to use it or not. This helps partially with the "Malicious Host" problem. Agents can decide if they trust the code they need in order to perform their goals. Of course again here you trust that the code is the original one and has not been modified but that doesn't give any guarantees that the platform will execute it correctly. Furthermore the agents are able to verify a host before they migrate to it. So if every host n can verify host n+1 then we can make sure that our agent moves in a selected path of hosts. If the host is not trusted then the agent may decide not to execute there. Of course the agent can select where to execute but it doesn't have any

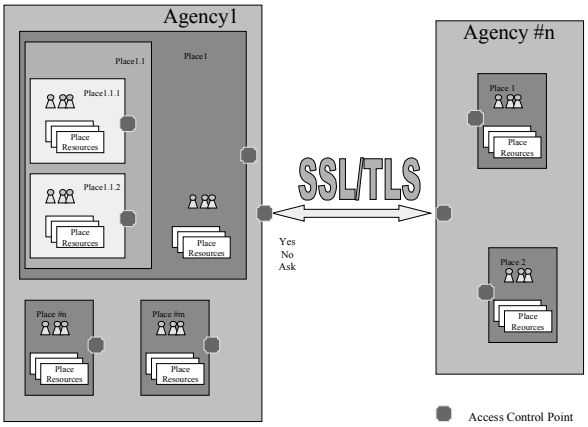

Figure 3 - Agent authentication/authorization route

guarantees after it arrives to that host, as its execution is controlled by that host's EE. User agents that are given permission can put their own code to this database and make it available to third party agents permanently or for a limited time. This increases the flexibility as well as the security and performance of the platform. The flexibility and performance because each user can have its own implementations of custom code on the node and thus his agents can be more lightweight and less complex. Security is also enhanced as the administrator will provide all new encryption/compression/etc algorithms with code he has tested and trusts. So agents don't bring every time their own code which in turn makes it less risky for the platform to be faced with unintentional side effects (e.g. buffer overflow). Not to mention that the administrator's implementations will be always updated and platform specific optimized, providing therefore better overall performance to the system.

The component database can be considered a general database of active code, protocols, encryption algorithms, etc. It can also be used for caching agent's code but its use is far more extended than simple caching. Furthermore this database can help with various matters that have to do with international law e.g. on exporting encryption algorithms. US Law export regulations force different policy on Java JCE APIs inside and outside USA. Therefore for non-USA users alternatively other implementations such as IAIK-JCE [10] could be provided. Also various algorithms are patented and their use should be allowed only to specific agents. These classes can be stored in the component database and via the right policy to be accessed only by the intended users.

Component database is of great significance to this approach as it ensures the up to date status of various components and also in parallel minimizes security risks for agents and for the platform. Security is by nature overhead in the communication and execution in order to protect the system. We accept that. Yet there are novel general ways/techniques to minimize this overhead (under certain conditions) and fortify the security on the node. In the future more specialized techniques that take optimal advantage of the underlying network resources could be used if this approach is to leave the research domain and enter the commercial one.

4.5 Resource Manager

A resource manager is available in order to handle the resources assigned to the agency or place. We assume that resources are assigned from the administrator (that is the person that creates the place and this can be the agency administrator or one of the previous n-1 place administrators who created the nested place n) to a place n and are managed by the owner of the newly created place. The resources and their management is transparent to place users and to nested places that place n might contain. The place resource manager can handle the resources that are dedicated to a specific place. It can be contacted also directly via the agents that reside in the associated place also in the case that there is a need for more resources.

Note that the resources available to a certain place are transparent to the agent and its users. That means that local resources could be extended via CORBA in order to access resources in other nodes. With this idea in mind one could consider network-wide working space and resource consumption (e.g. distributed disk space). This helps also with the Place Oriented Virtual Private Network (PO-VPN) [13]. In a PO-VPN scenario an enterprise can setup places spawned in a network infrastructure and therefore create a VPN of places where its agents can execute according to custom security policies and services. The transparency of resources across multiple agencies which host places that belong to a VPN or a 3rd party entity offers new hardly scratched ground for further interesting research.

4.6 Cache Manager

The cache (handled by the cache manager) is another essential part of the architecture and its usage is mainly focusing on improvement of the overall performance. Security checks are time and computing consuming processes. In our effort, not to duplicate all the time the necessary security checks, we have a cache. Security checks that have been done via the enforcement engine are stored with a time limit in the cache. If the time limit expires then the security checks are performed again, otherwise the security check is considered valid and is used by the system.

The policy DB can be dynamically updated via the enforcement engine any time. Thus the problem is faced that the cache contains outdated information. We solve this problem by deleting (each time the policy for an entity changes) the cached security checks that are associated with this key/person partially or completely. So next time that a security check is requested, it will not exist in cache and it will be performed from the beginning. This is a novel method to speed-up the performance of our system. The implementation of this approach requires modification of the JVM.

4.7 Audit Manager

Audit manager handles all audit events. Experience has shown that 100% security is difficult to realize - if not impossible - due to the multiple factors that interfere. Collecting data generated by network activity provides a useful tool in analyzing the existent security and also trace back (if possible) the originators of a security breakout. Having a detailed audit can lead to reconstruction of a sequence of events and better understanding of past security failures. Audit data include any attempt to achieve different security level or change entries in the system's databases etc. Intrusion attempts can also be detected via audit e.g. when we see repetitive failures in an attempt to use a component/service we can adapt our policy so that we prevent any possible intrusions. The more detailed the audit process is the better can various activities be debugged and protected from repeated errors or false configurations. Unfortunately not all activities can be monitored. Furthermore these logs are usually plain text files which introduces further security risks (acquirement of private info, alteration etc). Thus the log files should be protected with a computationally cheap method [9] which will make impossible for the attacker to read and also impossible to undetectably modify or destroy.

4.8 Enforcement Engine

The Enforcement Engine is used to enforce the policy on the agency in general and on the places. It is also the front-end environment via which users interact with the architecture. An Enforcement Engine must satisfy three important rules. It must be i) always invoked, ii) tamperproof iii) verifiable. We try to fulfill the above requirements by implicitly checking access rights to all systems resources, signing the components and loading the basic parts of the architecture securely. If a user gives (via the policy file or host's OS) write permission to everyone in his CLASSPATH directories, then another malicious user could alter the files of the enforcement engine or the policy to comfort his own goals. The host/agency/place administrator is able to use a GUI and edit the policy and credential data prior to system run. Changes can also be made dynamically during system runtime via agent interface. The enforcement engine we have heavily depends on Java's security architecture.

5. SUMMARY AND CONCLUSIONS

A security architecture for agent based systems has been presented. This defensive model of design is focused on designing agent systems to be secure from the scratch. Adding security after the design phase has been shown to be difficult, expensive and inadequate. Security is not an explicitly called service and its treatment as such imposes further security risks in the infrastructure. We tried to keep the architecture transparent and simple as it is easier to evolve and update it in order to cover future requirements.

We have showed that benefits such as simplicity, scalability, flexibility, interoperability, performance and safety have been addressed successfully. With the use of Java we can also guarantee a high level of safeness. The components of the architecture have been analyzed and explained.

Per identity/place security and customization as well as the rapid service creation is the main driving force for next generation mobile agent systems. Furthermore by combining modules in a Lego-like way (supported by the component DB) we believe that our approach tackles issues like survivability and interoperability.

In the future we intent to advance our approach. Our architecture tries to identify and prevent possible malicious agents. For the moment it can't handle collaborative attacks. Taking into account the tools provided (e.g. audit log, encryption tools, etc) one could implement stationary agents (guards) that reside on a place and based on intelligent internal strategy react to environment changes and try to track and eliminate collaborative attacks. Those guards could also work in collaboration thus providing a higher level of security to a number of hosts. As agent technology evolves and becomes more sophisticated a co-operative security infrastructure could be developed and deployed.

We understand also that this approach has its limits and is based also on 3rd party modules. Any security flow on Java is directly a security flaw also in the implementation of our approach since everything is based on Java. But with the evolvement of the language and in parallel the development and deployment of new algorithms and services we hope that the security architecture presented here will be able to fulfil its goals not only now but also in the future.

6. REFERENCES

[1] Cetus Links on Mobile Agents : http://www.cetus-links.org/oo_mobile_agents.html

[2] F. Hohl, Protecting mobile agents with Blackbox security, Proc. 1997 WS Mobile Agents and security, Univ. of Maryland.

[3] International Telecommunication Union, ITU-T Recommendation on X.509

[4] IAIK-SSL Implementation. URL : http://jcewww.iaik.tu-graz.ac.at/IAIK_JCE/jce.htm

[5] IETF Transport Layer Security (TLS) group http://www.consensus.com/ietf-tls/ietf-tls-home.html

[6] JASMIN URL : http://www.mrl.nyu.edu/meyer/jasmin/

[7] T. Sander and C. Tschudin. Protecting Mobile Agents against malicious hosts. Lecture Notes in Computer Science on Mobile Agent Security, November 1997.

[8] E. Gerck. "Overview of Certification Systems: X.509, CA, PGP and SKIP", Meta-Certificate Group, Novware Softex/Unicamp Brazil.

[9] B. Schneier and J. Kelsey, "Cryptographic Support for Secure Logs on Untrusted Machines", The 7th USENIX Security Symposium proceedings, USENIX Press, Jan 1998, pp. 53-62

[10] IAIK-JCE Implementation. URL : http://jcewww.iaik.tu-graz.ac.at/IAIK_JCE/jce.htm

[11] OMG Web Site : http://www.omg.org/

[12] FIPA Web Site: http://www.fipa.org/

[13] Stamatis Karnouskos, Ingo Busse, Stefan Covaci, "Place-Oriented Virtual Private Networks", HICSS-33, January 4-7 2000, on the island of Maui, Hawaii.

[14] Lightweight Directory Access Protocol (LDAP v3), RFC 2251. URL: http://info.internet.isi.edu/in-notes/rfc/files/rfc2251.txt

[15] Unified Modeling Language, Rational Software, URL : http://www.rational.com/uml

[16] "Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure", C. Ellison and B. Schneier, Computer Security Journal, v 16, n 1, 2000, pp. 1-7.

[17] Java Security Flaws http://kimera.cs.washington.edu/flaws/

[18] MASIF - Mobile Agent System Interoperability Facility, http://www.omg.org/docs/orbos/98-03-09.pdf