Countermeasures for Mobile Agent Security

Wayne A. Jansen
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
wjansen@nist.gov

Abstract: Security is an important issue for the widespread deployment of applications based on software agent technology. It is generally agreed that without the proper countermeasures in place, use of agent-based applications will be severely impeded. However, not all applications require the same set of countermeasures, nor can they depend entirely on the agent system to provide them. Instead, countermeasures are applied commensurate with the anticipated threat profile and intended security objectives for the application. While countermeasures typically include any action, device, procedure, technique, or other measure that reduces the vulnerability of or threat to a system, our focus here is specifically on technical mechanisms, as opposed to procedural or non-technical measures. Such countermeasures can be integrated directly into an agent system, or incorporated into the design of an agent to supplement the capabilities of an underlying agent system. This paper gives an overview of the threats associated with software agent systems and their technical countermeasures, including the strengths and weaknesses of the techniques involved. Emphasis is on mobile software agents, since they typically face a more severe set of threats than do static agents and, therefore, demand more rigorous countermeasures.

Keywords: Mobile Agents; Computer Security; Countermeasures

Introduction

Over the years computer systems have evolved from centralized monolithic computing devices supporting static applications, into networked environments that allow complex forms of distributed computing. A new phase of evolution is now under way based on software agents. A software agent is loosely defined as a program that can exercise an individual's or organization's authority, work autonomously toward a goal, and meet and interact with other agents. Possible interactions among agents include such things as contract and service negotiation, auctioning, and bartering.

Software agents may be either stationary or mobile. Stationary agents remain resident at a single platform, while mobile agents are capable of suspending activity on one platform and moving to another, where they resume execution. The concept of mobile code is not new, dating back to the 1960's when remote job entry systems were used to submit programs to a central computer. More recently code mobility has been popularized through the use of web browsers to download Java[1] applets from web servers. Mobile agents go one step further, allowing the complete mobility of software among supporting platforms to form large-scale, loosely-coupled distributed systems. The reader is referred to [34] for a more extensive introduction to the subject of software agents.

As the sophistication of mobile software increases, so too do the associated security threats and vulnerabilities. The focus of this paper is on the security issues that arise when agents are mobile, and the possible countermeasures for dealing with those issues. Various degrees of mobility exist, corresponding to the possible variations of relocating code and state information, including the values of instance variables, the program counter, execution stack, etc. [7]. For example, a simple agent written as a Java applet [9] has mobility of code through the movement of class files from a web server to a browser. However, no associated state information is conveyed. In contrast, Aglets [18], developed at IBM Japan, builds upon Java to allow the values of instance variables, but not the program counter or execution stack, to be conveyed along with the code as the agent relocates. For a stronger form of mobility, Sumatra [1], developed at the University of Maryland, allows Java threads to be conveyed along

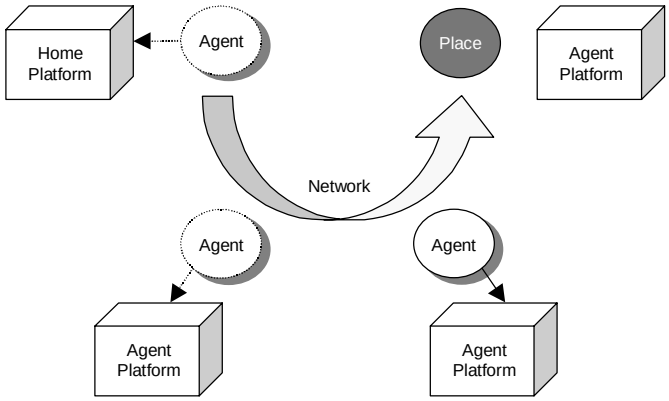
[1] Certain computer manufacturers' products and standards are discussed in this paper. The discussion is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the products and standards identified are necessarily the best available.

with the agent's code during relocation. The mobile agent systems under discussion here involve the relocation of both code and state information.

Security Threats

Threats to the security of mobile agents generally fall into four comprehensive classes: disclosure of information, denial of service, corruption of information, and interference or nuisance. We use the components of an agent system to further delineate threats by identifying the possible source and target of an attack with respect to elements within that paradigm. It is important to note that many of the threats that are discussed have counterparts in classical client-server systems and have always existed in some form in the past (e.g., executing any code from an unknown source either downloaded from a network or supplied on floppy disk). Mobile agents simply offer a greater opportunity for abuse and misuse, broadening the scale of threats significantly. New threats arising from the mobile agent paradigm are due to the fact that, contrary to the usual situation in computer security where the owner of the application and the operator of the computer system are the same, the agent's owner and system's operator are different.

A number of models exist for describing agent systems [2, 19, 34], however, for discussing security issues it is sufficient to use a very simple one, consisting of only two main components: the agent and the agent platform. An agent comprises the code and state information needed to carry out some computation. Multiple agents cooperate with one another to carry out some application. Mobility allows an agent to move or hop among agent platforms. The agent platform provides the computational environment in which an agent operates. The platform where an agent originates is referred to as the home platform, and normally is the most trusted environment for an agent. One or more hosts may comprise an agent platform, and an agent platform may support multiple locations or meeting places where agents can interact. Since some of these aspects do not affect the discussion of security issues, they are omitted from the agent system model illustrated in Figure 1, which depicts the movement of an



agent among several agent platforms.

Figure 1: Agent System Model

Four threat categories are identified: threats stemming from an agent attacking an agent platform, an agent platform attacking an agent, an agent attacking another agent on the agent platform, and other entities attacking the agent system. The cases of an agent attacking an agent on another agent platform and of an agent platform attacking another platform are covered within the last category, since these attacks are focused primarily on the communications capability of the platform to exploit potential vulnerabilities. The last category also includes more conventional attacks against the underlying operating system of the agent platform.

Agent Against Agent Platform

The mobile agent paradigm requires an agent platform to accept and execute code developed elsewhere. An incoming agent has two main lines of attack. The first is to gain unauthorized access to information residing at the

agent platform; the second is to use its authorized access in an unexpected and disruptive fashion. Unauthorized access may occur simply through a lack of adequate access control mechanisms at the platform or weak identification and authentication, which allows an agent to masquerade as one trusted by the platform. Once access is gained, information residing at the platform can be disclosed or altered. Besides confidential data, this information could include the instruction codes of the platform. Depending on the level of access, the agent may be able to completely shut down or terminate the agent platform. Even without gaining unauthorized access to resources, an agent can deny platform services to other agents by exhausting computational resources, if resource constraints are not established or not set tightly. Otherwise, the agent can merely interfere with the platform by issuing meaningless service requests wherever possible.

Agent Platform Against Agent

A receiving agent platform can easily isolate and capture an agent and may attack it by extracting information, corrupting or modifying its code or state, denying requested services, or simply reinitializing or terminating it completely. Extracting electronic cash directly from the agent is one simple example. An agent is very susceptible to the agent platform and may be corrupted merely by the platform responding falsely to requests for information or service, altering external communications, or delaying the agent until its task is no longer relevant. Extreme measures include the complete analysis and reversing engineering of the agent's design so that subtle changes can be introduced. Modification of the agent by the platform is a particularly insidious form of attack, since it can radically change the agent's behavior (e.g., turning a trusted agent into malicious one) or the accuracy of the computation (e.g., changing collected information to yield incorrect results).

Agent Against Other Agents

An agent can target another agent using several general approaches. These include actions to falsify transactions, eavesdrop upon conversations, or interfere with an agent's activity. For example, an attacking agent can respond incorrectly to direct requests it receives from a target or deny that a legitimate transaction occurred. An agent can gain information by serving as an intermediary to the target agent (e.g., through masquerade) or by using platform services to eavesdrop on intra-platform messages. If the agent platform has weak or no control mechanisms in place, an agent can access and modify another agent's data or code, or interfere with the agent by invoking its public methods (e.g., attempt buffer overflow, reset to initial state, etc.). Even with reasonable control mechanisms in place, an agent can attempt to send messages repeatedly to other agents in an attempt to deny them the ability to communicate.

Other Entities Against Agent System

Even assuming the locally active agents and the agent platform are well behaved, other entities both outside and inside the agent framework may attempt actions to disrupt, harm, or subvert the agent system. The obvious methods involve attacking the inter-agent and inter-platform communications through masquerade, (e.g., through forgery or replay) or intercept. For example, at a level of protocol below the agent-to-agent or platform-to-platform protocol, an entity may eavesdrop on messages in transit to and from a target agent or agent platform to gain information. An attacking entity may also intercept agents or messages in transit and modify their contents, substitute other contents, or simply replay the transmission dialogue at a later time in an attempt to disrupt the synchronization or integrity of the agent framework. Denial of service attacks through available network interfaces is another possibility.

Countermeasures

Countermeasures refer to any action, device, procedure, technique, or other measure that reduces the vulnerability of or threat to a system. Many conventional security techniques used in contemporary distributed applications (e.g., client-server) also have utility as countermeasures within the mobile agent paradigm. Moreover, there are a number of extensions to conventional techniques and techniques devised specifically for controlling mobile code and executable content (e.g., Java applets) that are applicable to mobile agent security. We review these

countermeasures by considering those techniques that can be used to protect agent platforms, separately from those used to protect the agents that run on them.

Most agent systems rely on a common set of baseline assumptions regarding security. The first is that an agent trusts the home platform where it is instantiated and begins execution. The second is that the home platform and other equally trusted platforms are implemented securely, with no flaws or trapdoors that can be exploited, and behave non-maliciously. The third is that public key cryptography, primarily in the form of digital signature, is utilized through certificates and revocation lists managed through a public key infrastructure.

Protecting the Agent Platform

Without adequate defenses, an agent platform is vulnerable to attack from many sources including malicious agents, malicious platforms, and other malicious entities. Fortunately most conventional protection techniques, traditionally employed in trusted systems and communication security, can be used to provide analogous protection mechanisms for the agent platform. This is due in large part to the traditional role hardware plays as the foundation upon which software protection mechanisms are built. That is, within the mobile agent paradigm, the agent platform is a counterpart to a trusted host within the traditional framework.

One of the main concerns with an agent system implementation is ensuring that agents are not able to interfere with one another or with the underlying agent platform. One common approach for accomplishing this is to establish separate isolated domains for each agent and the platform, and control all inter-domain access. In traditional terms this concept, as it applies to a trusted computing base, is referred to as a reference monitor [31]. An implementation of a reference monitor has the following characteristics:

- It is always invoked and non-bypassable, mediating all accesses;
- It is tamper-proof; and
- It is small enough to be analyzed and tested.

Implementations of the reference monitor concept have been around since the early 1980's and employ a number of conventional security techniques, which are applicable to the agent environment. Such conventional techniques include the following:

- Mechanisms to isolate processes from one another and from the control process,
- Mechanisms to control access to computational resources,
- Cryptographic methods to encipher information exchanges,
- Cryptographic methods to identify and authenticate users, agent, and platforms, and
- Mechanisms to audit security relevant events occurring at the agent platform.

More recently developed techniques aimed at mobile code and mobile agent security have for the most part evolved along these traditional lines. Techniques devised for protecting the agent platform are described in the following sections.

Software-Based Fault Isolation

Software-Based Fault Isolation [33], as its name implies, is a method of isolating application modules into distinct fault domains enforced by software. The technique allows untrusted programs written in an unsafe language, such as C, to be executed safely within the single virtual address space of an application. Untrusted machine interpretable code modules are transformed so that all memory accesses are confined to code and data segments within their fault domain. Access to system resources can also be controlled through a unique identifier associated with each domain. The technique, commonly referred to as sandboxing, is highly efficient compared with using hardware page tables to maintain separate address spaces for modules, when the modules communicate frequently among fault domains. It is also ideally suited for situations where most of the code falls into one domain that is trusted, since modules in trusted domains incur no execution overhead.

Safe Code Interpretation

Agent systems are often developed using an interpreted script or programming language. The main motivation for doing this is to support agent platforms on heterogeneous computer systems. Moreover, the higher conceptual level of abstraction provided by an interpretative environment can facilitate the development of the agent's code [24]. The idea behind Safe Code Interpretation is that commands considered harmful can be either made safe for, or denied to, an agent. For example, a good candidate for denial would be the command to execute an arbitrary string of data as a program segment.

One of the most widely used interpretative languages today is Java. The Java programming language and runtime environment [8, 9] enforces security primarily through strong type safety. Java follows a so-called sandbox security model, used to isolate memory and method access, and maintain mutually exclusive execution domains. Security is enforced through a variety of mechanisms. Static type checking in the form of byte code verification is used to check the safety of downloaded code. Some dynamic checking is also performed during runtime. A distinct name space is maintained for untrusted downloaded code and linking of references between modules in different name spaces is restricted to public methods. A security manager mediates all accesses to system resources, serving in effect as a reference monitor. In addition, Java inherently supports code mobility, dynamic code downloading, digitally signed code, remote method invocation, object serialization, platform heterogeneity, and other features that make it an ideal foundation for agent development. There are many agent systems based on Java, including Aglets [16, 18], Mole [29], Ajanta [17], and Voyager [21]. However, limitations of Java to account for memory, CPU, and network resources consumed by individual threads [5] and to support thread mobility [7] have been noted.

Probably the best known of the safe interpreters for script-based languages is Safe Tcl [23], which was used in the early development of the Agent Tcl [10] system. Safe Tcl employs a padded cell concept, whereby a second "safe" interpreter pre-screens any harmful commands from being executed by the main Tcl interpreter. The term padded cell refers to this isolation and access control technique, which provides the foundation for implementing the reference monitor concept. More than one safe interpreter can be used to implement different security policies, if needed. Constructing policy-based interpreters requires skill to avoid overly restrictive or unduly protected computing environments.

In general, current script-based languages do not incorporate an explicit security model in their design, and rely mainly on decisions taken during implementation. One recent exception is the implementation of secure JavaScript in Mozilla [3]. The implementation uses a padded cell to control access to resources and external interfaces, prevents residual information from being retained and accessible among different contexts operating simultaneously or sequentially, and allows policy, which partitions the name space for access control purposes, to be specified independently of mechanism.

Signed Code

A fundamental technique for protecting an agent system is signing code or other objects with a digital signature. A digital signature serves as a means of confirming the authenticity of an object, its origin, and its integrity. Typically the code signer is either the creator of the agent, the user of the agent, or some entity that has reviewed the agent. Because an agent operates on behalf of an end-user or organization, mobile agent systems [10, 16, 17, 30] commonly use the signature of the user as an indication of the authority under which the agent operates.

Code signing involves public key cryptography, which relies on a pair of keys associated with an entity. One key is kept private by the entity and the other is made publicly available. Digital signatures benefit greatly from the availability of a public key infrastructure, since certificates containing the identity of an entity and its public key (i.e., a public key certificate) can be readily located and verified. Passing the agent's code through a non-reversible hash function, which provides a fingerprint or unique message digest of the code, and then encrypting the result with the private key of the signer forms a digital signature. Because the message digest is unique, and thus bound to the code, the resulting signature also serves as an integrity mechanism. The agent code, signature, and public key certificate can then be forwarded to a recipient, who can easily verify the source and authenticity of the code.

Note that the meaning of a signature may be different depending on the policy associated with the signature scheme and the party who signs. For example, the author of the agent, either an individual or organization, may

use a digital signature to indicate who produced the code, but not to guarantee that the agent performs without fault or error. In fact, author-oriented signature schemes were originally intended to serve as digital shrink wrap, whereby the original product warranty limitations stated in the license remain in effect (e.g., manufacturer makes no warranties as to the fitness of the product for any particular purpose).

Microsoft's Authenticode, a common form of code signing, enables Java applets or Active X controls to be signed, ensuring users that the software has not been tampered with or modified and that the identity of the author is verified. For many users, however, the signature has gone beyond establishing authenticity and become a form of trust in the software, which could ultimately have disastrous consequences.

Rather than relying solely on the reputation of a code producer, it would be prudent to have an independent review and verification of code performed by a trusted party or rating service [13]. For example, the decision to execute an agent may be taken only with the endorsement, in the form of a digital signature over the code, by a site security administrator.

State Appraisal

The goal of State Appraisal [6] is to ensure that an agent has not been somehow subverted due to alterations of its state information. The success of the technique relies on the extent to which harmful alterations to an agent's state can be predicted, and countermeasures, in the form of appraisal functions, can be prepared before using the agent. Appraisal functions are used to determine what privileges to grant an agent, based both on conditional factors and whether identified state invariants hold. An agent whose state violates an invariant can be granted no privileges, while an agent whose state fails to meet some conditional factors may be granted a restricted set of privileges.

Both the author and owner of an agent produce appraisal functions that become part of an agent's code. An owner typically applies state constraints to reduce liability and/or control costs. When the author and owner each digitally sign the agent, their respective appraisal functions are protected from undetectable modification. An agent platform uses the functions to verify the correct state of an incoming agent and to determine what privileges the agent can possess during execution. Privileges are issued by a platform based on the results of the appraisal function and the platform's security policy. It is not clear how well the theory will hold up in practice, since the state space for an agent could be quite large, and while appraisal functions for obvious attacks may be easily formulated, more subtle attacks may be significantly harder to foresee and detect. The developers of the technique, in fact, indicate it may not always be possible to distinguish normal results from deceptive alternatives.

Path Histories

The basic idea behind Path Histories [4, 22] is to maintain an authenticatable record of the prior platforms visited by an agent, so that a newly visited platform can determine whether to process the agent and what resource constraints to apply. Computing a path history requires each agent platform to add a signed entry to the path, indicating its identity and the identity of the next platform to be visited, and to supply the complete path history to the next platform. To prevent tampering, the signature of the new path entry must include the previous entry in the computation of the message digest. Upon receipt, the next platform can determine whether it trusts the previous agent platforms that the agent visited, either by simply reviewing the list of identities provided or by individually authenticating the signatures of each entry in the path history to confirm identity. While the technique does not prevent a platform from behaving maliciously, it serves as a deterrent, since the platform's signed path entry is non-repudiatable. One obvious drawback is that path verification becomes more costly as the path history increases. The technique is also dependent on the ability of a platform to judge correctly whether to trust the visited platforms identified.

Proof Carrying Code

The approach taken by Proof Carrying Code [20], obligates the code producer (e.g., the author of an agent) to formally prove that the program possesses safety properties previously stipulated by the code consumer (e.g., security policy of the agent platform). The aim of the technique is to prevent the execution of unsafe code. The code and proof are sent together to the code consumer where the safety properties can be verified. A safety

predicate, representing the semantics of the program, is generated directly from the native code to ensure that the companion proof does in fact correspond to the code. The proof is structured in a way that makes it straightforward to verify without using cryptographic techniques or external assistance. Once verified, the code can run without further checking. Any attempts to tamper with either the code or the safety proof result in either a verification error or, if the verification succeeds, a safe code transformation.

Initial research has demonstrated the applicability of Proof Carrying Code for fine-grained memory safety, and shown the potential for other types of safety policies, such as controlling resource use. Thus, the technique could serve as a reasonable alternative to Software-Based Fault Isolation in some applications. The theoretical underpinnings of Proof Carrying Code are based on well-established principles from logic, type theory, and formal verification. There are, however, some potentially difficult problems to solve before the approach is considered practical. They include a standard formalism for establishing security policy, automated assistance for the generation of proofs, and techniques for limiting the potentially large size of proofs that in theory can arise. In addition, the technique is tied to the hardware and operating environment of the code consumer, which may limit its applicability.

Protecting Agents

While countermeasures directed toward platform protection are a direct evolution of traditional mechanisms employed by trusted hosts, those directed toward agent protection often depart somewhat more radically from traditional lines. This is due to the fact that traditional mechanisms were not devised to address threats stemming from attacks on the application by the execution environment, which is precisely the situation faced by an agent executing of an agent platform that it may not completely trust.

The problem stems from the inability to effectively extend the trusted environment of an agent's home platform to other agent platforms visited by the agent. For some applications, the threat profile may not demand complex protection schemes. For example, agents that do not accumulate state, or convey their results back to the home platform after each hop, have less exposure to certain attacks. For other applications, basic protection schemes may prove adequate. For example, the JumpingBeans [14] agent system originally addressed mobile agent security issues by implementing a client-server architecture, whereby an agent always returns to a secure, trusted central host first, before moving onto any other platform. Agent systems that allow for more decentralized mobility, such as IBM Aglets, prevent the receiving platform from accepting agents from an agent platform that is not defined as a trusted peer within the receiving platform's security policy. Alternatively, the originator can restrict an agent's itinerary to only a trusted set of platforms known in advance.

While these basic schemes may have value in specific situations, they do not support the free-roaming, autonomous capabilities envisioned for many agent applications. More general-purpose techniques for protecting an agent are discussed in the following sections.

Partial Result Encapsulation

One approach used to detect tampering by malicious hosts is to encapsulate the results of an agent's actions, at each platform visited, for subsequent verification, either when the agent returns to the point of origination or possibly at intermediate points as well. Encapsulation may be done for different purposes with different mechanisms, such as providing confidentiality using encryption, or providing integrity and accountability using digital signature. The information encapsulated depends somewhat on the goals of the agent, but typically includes responses to inquiries made or transactions issued at the platform. In general, there are three alternative ways to encapsulate partial results:

- Provide the agent with a means for encapsulating the information,
- Rely on the encapsulation capabilities of the agent platform, or
- Rely on a trusted third party to timestamp a digital fingerprint of the results.

While none of the alternatives prevents malicious behavior by the platform, they do allow certain types of tampering to be detected. One difference with an agent-controlled encapsulation is that it can be applied

independently in the design of an appropriate agent application, regardless of the capabilities of the agent platform or supporting infrastructure.

Often the amount of information gathered by an agent is rather small in comparison to the size of the encryption keys involved and the resulting ciphertext. A solution called sliding encryption [36] allows small amounts of data to be encrypted and yield efficient sized results. The scenario envisioned for use of sliding encryption is one in which the agent, using a public key it carries, encrypts the information as it is accumulated at each platform visited. Later, when the agent returns to the point of origination, the information is decrypted using the private key maintained there. While the purpose of sliding encryption is confidentiality, an additional integrity measure could be applied as well, before encryption occurs.

Another method for an agent to encapsulate result information is to use Partial Result Authentication Codes (PRAC) [35], which are cryptographic checksums formed using secret key cryptography (i.e., message authentication codes). This technique requires the agent and its originator to maintain or incrementally generate a list of secret keys used in the PRAC computation. Once a key is applied to encapsulate the information collected, the agent destroys it before moving onto the next platform, guaranteeing forward integrity. The forward integrity property ensures that if one of the servers visited is malicious, the previous set of partial results remains valid. However, only the originator can verify the results, since no other copies of the secret key remain. As an alternative, public key cryptography and digital signatures can be used in lieu of secret key techniques. One benefit is that authentication of the results can be made a publicly verifiable process at any platform along the way, while maintaining forward integrity.

The PRAC technique has a number of limitations. The most serious occurs when a malicious platform retains copies of the original keys or key generating functions of an agent. If the agent revisits the platform or visits another platform conspiring with it, a previous partial result entry or series of entries could be modified without the possibility of detection. Since the PRAC is oriented toward integrity and not confidentiality, the accumulated set of partial results can also be viewed by any platform visited, although this is easily resolved by applying sliding key or other forms of encryption.

Rather than relying on the agent to encapsulate the information, each platform can be required to encapsulate partial results along the way [4]. The distinction is not only one of where the encapsulation mechanisms are retained, either with the agent or at a platform, but also one of responsibility and associated liabilities. While it appears rather straightforward to change from one orientation to the other, the nuances can be significant. For example, an agent that used to sign partial results with the public key of its originator must be revised to have all platforms it visits sign the results with their private key. If the agent triggers a digital signature encapsulation via a platform programming interface, the implementation must ensure that enough context is included (e.g., the time and query issued) so that the agent cannot induce the platform to sign arbitrary information.

Karjoth and his associates [15] devised a platform-oriented technique for encapsulating partial results, which reformulates and improves upon the PRAC technique. The approach is to construct a chain of encapsulated results that binds each result entry to all previous entries and to the identity of the subsequent platform to be visited. Each platform digitally signs its entry using its private key, and uses a secure hash function to link results and identities within an entry. Besides forward integrity, the encapsulation technique also provides confidentiality by encrypting each piece of accumulated information with the public key of the originator of the agent. Moreover, the forward integrity is stronger than that achieved with PRAC, since a platform is unable to modify its entry in the chain, should it be revisited by the agent, or to collude with another platform to modify entries, without invalidating the chain. A variant of this technique, which uses message authentication codes in lieu of digital signatures, is also described.

Yee, who proposed the PRAC technique, noted that forward integrity could also be achieved using a trusted third party that performs digital timestamping. A digital timestamp [11] allows one to verify that the contents of a file or document existed as such, at a particular point in time. Yee raises the concern that the granularity of the timestamps may limit an agent's maximum rate of travel, since it must reside at one platform until the next time period. Another possible concern is the general availability of a trusted timestamping infrastructure.

Mutual Itinerary Recording

One interesting variation of Path Histories is a general scheme for allowing an agent's itinerary to be recorded and tracked by another cooperating agent and vice-versa [26], in a mutually supportive arrangement. When moving between agent platforms, an agent conveys the last platform, current platform, and next platform information to the cooperating peer through an authenticated channel. The peer maintains a record of the itinerary and takes appropriate action when inconsistencies are noted. Attention is paid so that an agent avoids platforms already visited by its peer. The rationale behind this scheme is founded on the assumption that only a few agent platforms are malicious, and even if an agent encounters one, the platform is not likely to collaborate with another malicious platform being visited by the peer. Therefore, by dividing up the operations of the application between two agents, certain malicious behavior of an agent platform can be detected.

The scheme can be generalized to more than two cooperating agents. For some applications it is also possible for one of the agents to remain static at the home platform. Because the path records are maintained at the agent level, this technique can be incorporated into any appropriate application. Some drawbacks mentioned include the cost of setting up the authenticated channel and the inability of the peer to determine which of the two platforms is responsible if the agent is killed.

Itinerary Recording with Replication and Voting

A faulty agent platform can behave similarly to a malicious one. Therefore, applying fault tolerant capabilities to this environment should help counter the effects of malicious platforms. One such technique for ensuring that a mobile agent arrives safely at its destination is through the use of replication and voting [28]. The idea is that rather than using a single copy of an agent to perform a computation, multiple copies are used. Although a malicious platform may corrupt a few copies of the agent, enough replicas avoid the encounter to successfully complete the computation.

For each stage of the computation, the platform ensures that arriving agents are intact, carrying valid credentials. Platforms involved in a particular stage of a computation are expected to know the set of acceptable platforms for the previous stage. Based on the inputs it receives, the platform propagates onto the next stage only a subset of the replica agents it considers valid. Underlying assumptions are that the majority of the platforms are well behaved, and most agents arrived unscathed with equivalent results at each stage. One of the protocols used in this technique requires forwarded agents to convey the accumulated signature entries of all forwarding nodes onto each stage. The approach taken is similar to Path Histories, but extended with fault tolerant capabilities. The technique seems appropriate for specialized applications where agents can be duplicated without problems, the task can be formulated as a multi-staged computation, and survivability is a major concern. One obvious drawback is the additional resources consumed by replicate agents.

Execution Tracing

Execution tracing [32] is a technique for detecting unauthorized modifications of an agent through the faithful recording of the agent's behavior during its execution on each agent platform. The technique requires each platform involved to create and retain a non-repudiatable log or trace of the operations performed by the agent while resident there, and to submit a cryptographic hash of the trace upon conclusion as a trace summary or fingerprint. A trace is composed of a sequence of statement identifiers and platform signature information. The signature of the platform is needed only for those instructions that depend on interactions with the computational environment maintained by the platform. For instructions that rely only on the values of internal variables, a signature is not required and, therefore, is omitted. The technique also defines a secure protocol to convey agents and associated security related information among the various parties involved, which may include a trusted third party to retain the sequence of trace summaries for the agent's entire itinerary. If any suspicious results occur, the appropriate traces and trace summaries can be obtained and verified, and a malicious host identified.

The approach has a number of drawbacks, the most obvious being the size and number of logs to be retained, and the fact that the detection process is triggered sporadically, based on suspicious results or other factors. Other more subtle problems identified include the lack of accommodating multi-threaded agents and dynamic optimization

techniques. While the goal of the technique is to protect an agent, the technique does afford some protection for the agent platform, providing that the platform can also obtain the relevant trace summaries and traces from the various parties involved.

Environmental Key Generation

Environmental Key Generation [25] describes a scheme for allowing an agent to take predefined action when some environmental condition is true. The approach centers on constructing agents in such a way that upon encountering an environmental condition (e.g., via a matched search string), a key is generated, which is used to unlock some executable code cryptographically. The environmental condition is hidden through either a one-way hash or public key encryption of the environmental trigger. The technique ensures that a platform or an observer of the agent cannot uncover the triggering message or response action by directly reading the agent's code.

The procedure is somewhat akin to the way in which modern operating systems apply passwords to determine whether login attempts are valid (i.e., the password is used as a cryptographic key). One weakness of this approach is that a platform, which completely controls the agent, could simply modify the agent to print out the unlocked executable code upon receipt of the trigger, instead of executing it. Another drawback is that an agent platform typically limits the capability of an agent to execute code created dynamically, since it is considered an unsafe operation. An author of an agent can apply the technique, however, in conjunction with other protection mechanisms for specific applications on appropriate platforms.

Computing with Encrypted Functions

The goal of Computing with Encrypting Functions [27] is to determine a method whereby mobile code can safely compute cryptographic primitives, such as a digital signature, even though the code is executed in untrusted computing environments and operates autonomously without interactions with the home platform. The approach is to have the agent platform execute a program embodying an enciphered function without being able to discern the original function; the approach requires differentiation between a function and a program that implements the function.

For example, Alice has an algorithm to compute a function f. Bob has input x and wants to compute f(x) for Alice, but she doesn't want Bob to learn anything about f. If f can be encrypted in a way that results in another function E(f), then Alice can create a program P(E(f)), which implements E(f), and send it to Bob, embedded within her agent. Bob then runs the agent, which executes P(E(f)) on x, and returns the result to Alice who decrypts it to obtain f(x). If f is a signature algorithm with an embedded key, the agent has an effective means to sign information without the platform discovering the key. Similarly, if it is an encryption algorithm containing an embedded key, the agent has an effective means to encrypt information at the platform.

Although the idea is straightforward, the trick is to find appropriate encryption schemes that can transform arbitrary functions as intended. The technique has been shown to be useful to encrypt rationale functions, and it remains to be seen whether more general functions can be encrypted in a similar fashion. The technique, while very powerful, does not prevent denial of service, replay, experimental extraction, and other forms of attack against the agent.

Obfuscated Code

Hohl [12] gives a detailed overview of the threats stemming from an agent encountering a malicious host as motivation for Blackbox Security. The strategy behind this technique is simple -- scramble the code in such a way that no one is able to gain a complete understanding of its function (i.e., specification and data), or to modify the resulting code without detection. A serious problem with the general technique is that there is no known algorithm or approach for providing Blackbox protection. However, the paper cites Computing with Encrypted Functions as an example of an approach that falls into the Blackbox category, with certain reservations concerning the limited range of input specifications that apply.

A time limited variation of Blackbox protection is introduced as a reasonable alternative, whereby the strength of the scrambling does not necessarily imply encryption as with the unqualified one, but relies mainly on obfuscation algorithms. Since an agent can become invalid before completing its computation, obfuscated code is suitable for applications that do not convey information intended for long-lived concealment. The examples given for pure obfuscation algorithms seem rather trivial and potentially ineffective. One promising method relies on a trusted host to trigger the execution of an agent's code segment. It is not strictly speaking a pure obfuscation algorithm, however, since code is redistributed to a trusted host, which is not part of the originally proposed scheme. The method does suggest a possible relationship between Environmental Key Generation and Obfuscated Code techniques. A serious drawback to this technique is the lack of an approach for quantifying the protection interval provided by the obfuscation algorithm, thus making it difficult to apply in practice. Furthermore, no techniques are currently known for establishing the lower bounds on the complexity for an attacker to reverse engineer an agent's code.

Summary

This paper identifies a range of measures for countering identified threats and fulfilling the security objectives of some, but not all, agent-based applications. While a wide variety of techniques do exist, not all are compatible with one another, due to their redundancy in purpose and overlap in functionality. For example, Path Histories and its public key cryptography underpinnings would be unnecessary in protecting an agent platform, if adequate proofs for an agent could be formulated to meet the platform's policy using the Proof Carrying Code technique. Similarly, Computing with Encrypted Functions conceptually overshadows Execution Tracing in protecting the agent. A useful way of viewing countermeasures in this regard is to differentiate between techniques oriented toward detection and those oriented toward prevention. Detection implies that the technique is aimed at discovering unauthorized modification of code or state information. Prevention implies that the technique is aimed at keeping the code and state information from being affected in a meaningful but negative way, as opposed to random alterations. To be effective, detection techniques are more likely than prevention techniques to depend on a legal or other social framework to penalize misbehavior.

The distinction between detection and prevention can be a bit arbitrary at times, since prevention often implicitly involves detection. Nevertheless, a determination is possible. Table 1 provides a categorization of the countermeasures reviewed. All other things being equivalent, a prevention technique is preferable to one oriented toward detection. However, many characteristics influence the choice of countermeasures. Such characteristics include the scope of protection, when and how the countermeasure is applied, dependencies, permanency, and the degree of difficulty in applying it.

Table 1: Categorization of Countermeasures

Countermeasure	Category	Security Objective
Signed Code	Detection	Agent Platform
State Appraisal	Detection	Agent Platform
Path Histories	Detection	Agent Platform
Partial Result Encapsulation	Detection	Agent
Mutual Itinerary Recording	Detection	Agent
Itinerary Recording with Replication and Voting	Detection	Agent
Execution Tracing	Detection	Agent
Software-Based Fault Isolation	Prevention	Agent Platform
Safe Code Interpretation	Prevention	Agent Platform
Proof Carrying Code	Prevention	Agent Platform
Environmental Key Generation	Prevention	Agent
Computing with Encrypted Functions	Prevention	Agent
Obfuscated Code	Prevention	Agent

No application is expected to benefit from wholesale adoption of the available techniques, simply due to the overhead and complexity involved. In order to be effectively applied, many of these techniques must be embodied with an agent system, while others can be applied independently within the context of a particular application. An overall framework that integrates these techniques into an effective security model is clearly needed.

Conclusions

The area of mobile agent security is in a state of immaturity, but rapidly improving. The traditional orientation toward host-based security persists and, therefore, available protection mechanisms tend to focus on protecting the agent platform. Emphasis is beginning to move toward developing techniques that are oriented toward protecting the agent, a much more difficult problem. The initial efforts are encouraging and will hopefully continue. There are a number of agent-based application domains for which basic and conventional security techniques should prove adequate. The countermeasures reviewed in this paper complement those techniques to reach a wider range of application domains. However, applications are expected to require a more comprehensive set of mechanisms and a flexible framework in which to apply a subset of selected mechanisms to meet their needs. In many ways, the success of software agent technology in general, and mobile agent technology in particular, depends on how well those needs are met.

Whether this evolutionary branch of computing successfully propagates forward or eventually dies out, remains to be seen. Nevertheless, mobile agent technology is intriguing and offers a new paradigm for application development that cannot be summarily dismissed. In addition, the security techniques being devised as countermeasures for this environment are expected to have applicability within other contexts as well.

References

[1] A. Acharya, M. Ranganathan, J. Salz, Sumatra: A Language for Resource-aware Mobile Programs, in J. Vitek and C. Tschudin (Eds.), Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, Lecture Notes in Computer Science No. 1222, April 1997, pp. 111-130.
<URL: http://www.cs.umd.edu/~acha/papers/lncs97-1.html>

[2] Agent Management, FIPA '97 Specification, part 1, version 2.0, Foundation for Intelligent Physical Agents, October 1998.
<URL: http://www.fipa.org/spec/FIPA97.html>

[3] V. Anupam, A. Mayer, Secure Web Scripting, IEEE Internet Computing, vol. 2, no. 6, November/December 1998, pp. 46-55.

[4] D. Chess et ali., Itinerant Agents for Mobile Computing, IEEE Personal Communications, vol. 2, no. 5, October 1995, pp. 34-49.

[5] G. Czajkowski, T. von Eicken, JRes: A Resource Accounting Interface for Java, ACM Conference on Object Oriented Languages and Systems (OOPSLA), Vancouver, Canada, October 1998.

[6] W. Farmer, J. Guttman, V. Swarup, Security for Mobile Agents: Authentication and State Appraisal, Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS '96), September 1996, pp. 118-130.

[7] A. Fuggetta, G. P. Picco, and G. Vigna, Understanding Code Mobility, IEEE Transactions on Software Engineering, 24(5), May 1998, pp. 342-361.
<URL: http://www.cs.ucsb.edu/~vigna/listpub.html>

[8] L. Gong, Java Security Architecture (JDK 1.2), Draft Document, revision 0.8, Sun Microsystems, March 1998.
<URL: http://service.felk.cvut.cz/doc/java/share/jdk1.2beta3/docs/guide/security/spec/security-spec.doc.html>

[9] J. Gosling, H. McGilton, The Java Language Environment: A White Paper, Sun Microsystems, May 1996.
<URL: http://SunSITE.sut.ac.jp/java/whitepaper/>

[10] R. S. Gray, Agent Tcl: A Flexible and Secure Mobile-Agent System, Proceedings of the 4th Annual Tcl/Tk Workshop (TCL 96), July 1996, pp. 9-23.
<URL: http://actcomm.dartmouth.edu/papers/#security>

[11] S. Haber, W. S. Stornetta, How to Time-Stamp a Digital Document, Journal of Cryptology, vol. 3, 1991, pp. 99-111.

[12] F. Hohl, Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts, in G. Vinga (Ed.), Mobile Agents and Security, Springer-Verlag, Lecture Notes in Computer Science No. 1419, 1998, pp. 92-113.
<URL: http://mole.informatik.uni-stuttgart.de/papers.html>

[13] N. Islam, et ali., A Flexible Security System for Using Internet Content, IEEE Software, September 1997, pp. 52-59.

[14] Jumping Beans White Paper, Ad Astra Engineering Inc., Sunnyvale CA, December 1998.
<URL: http://www.jumpingbeans.com/>

[15] G. Karjoth, N. Asokan, C. Gülcü, Protecting the Computation Results of Free-Roaming Agents, Proceedings of the Second International Workshop on Mobile Agents, Stuttgart, Germany, September 1998.

[16] G. Karjoth, D. B. Lange, M. Oshima, A Security Model for Aglets, IEEE Internet Computing, August 1997, pp. 68-77.

[17] N. Karnik, Security in Mobile Agent Systems, Ph.D. Dissertation, Department of Computer Science, University of Minnesota, October 1998.
<URL: http://www.cs.umn.edu/Ajanta/>

[18] D. B. Lange, M. Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.

[19] Mobile Agent System Interoperability Facilities Specification, Object Management Group (OMG) Technical Committee (TC) Document orbos/97-10-05, November 1997.
<URL: http://www.omg.org/techprocess/meetings/schedule/Technology_Adoptions.html#tbl_MOF_Specification>

[20] G. Necula, P. Lee, Safe Kernel Extensions without Run-Time Checking, Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI '96), Seattle, Washington, October 1996, pp. 229-243.
<URL: http://www.cs.cmu.edu/~necula/papers.html>

[21] ObjectSpace Voyager Core Package Technical Overview, version 1.0, ObjectSpace Inc., December 1997
<URL: http://www.objectspace.com/>

[22] J. J. Ordille, When Agents Roam, Who Can You Trust? Proceedings of the First Conference on Emerging Technologies and Applications in Communications, Portland, Oregon, May 1996
<URL: http://plan9.bell-labs.com/cm/cs/who/joann/ordille2.html>

[23] J. K. Ousterhout, J. Y. Levy, B. B. Welch, The Safe-Tcl Security Model, Technical Report SMLI TR-97-60, Sun Microsystems, 1997.

[24] J. K. Ousterhout, Scripting: Higher-Level Programming for the 21st Century, IEEE Computer, March 1998, pp. 23-30.

[25] J. Riordan, B. Schneier, Environmental Key Generation Towards Clueless Agents, in G. Vinga (Ed.), Mobile Agents and Security, Springer-Verlag, Lecture Notes in Computer Science No.1419, 1998.

[26] V. Roth, Secure Recording of Itineraries Through Cooperating Agents, Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, INRIA, France, 1998, pp. 147-154.

[27] T. Sander, C. Tschudin, Protecting Mobile Agents Against Malicious Hosts, in G. Vinga (Ed.), Mobile Agents and Security, Springer-Verlag, Lecture Notes in Computer Science No.1419, 1998, pp. 44-60.
<URL: http://www.icsi.berkeley.edu/~tschudin/>

[28] F. B. Schneider, Towards Fault-Tolerant and Secure Agentry, Proceedings 11th International Workshop on Distributed Algorithms, Saarbucken, Germany, September 1997.

[29] M. Straßer, J. Baumann, F. Hohl, Mole - A Java Based Mobile Agent System, in M. Mühlhäuser (Ed.), Special Issues in Object Oriented Programming, Verlag, 1997, pp. 301-308.
<URL: http://mole.informatik.uni-stuttgart.de/papers.html>

[30] J. Tardo, L. Valente, Mobile Agent Security and Telescript, Proceedings of IEEE COMPCON '96, Santa Clara, California, February 1996, IEEE Computer Society Press, pp. 58-63.

[31] Trusted Computer System Evaluation Criteria, Department of Defense, CSC-STD-001-83, Library No. S225 711, August 1983.
<URL: http://www.cru.fr/securite/Documents-generaux/orange.book>

[32] G. Vigna, Protecting Mobile Agents Through Tracing, Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems, Jyvälskylä, Finland, June 1997.
<URL: http://www.cs.ucsb.edu/~vigna/listpub.html>

[33] R. Wahbe, S. Lucco, T. Anderson, Efficient Software-Based Fault Isolation, Proceedings of the 14th ACM Symposium on Operating Systems Principles, ACM SIGOPS Operating Systems Review, December 1993, pp. 203-216.
<URL: http://www.cs.washington.edu/homes/tom/ >

[34] J. E. White, Mobile Agents, in J. M. Bradshaw (Ed.) Software Agents, AAAI/The MIT Press, 1997.

[35] B. S. Yee, A Sanctuary for Mobile Agents, Technical Report CS97-537, University of California in San Diego, April 1997.
<URL: http://www-cse.ucsd.edu/users/bsy/index.html>

[36] A. Young, M. Yung, Sliding Encryption: A Cryptographic Tool for Mobile Agents, Proceedings of the 4th International Workshop on Fast Software Encryption, FSE '97, January 1997.