Security Mechanisms for Mobile Agent Based Vulnerability Analysis Tools

A Partial Review of Techniques

Rich Goyette

28 Nov 2002

This paper was prepared for Professor A. Karmouch in partial fulfillment of the requirements for the course ELG7187A: Mobile Software Agent Technologies for Telecommunications - Fall 2002.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

1 Introduction

Mobile agent technology has become one of the most active areas of research in the field of information technology [1][2]. Truly mobile, autonomous agents promise several advantages over current network models and paradigms. For example, mobile agents do their work remotely thereby reducing issues related to network latency and bandwidth [3]. They promise to be efficient and personal allowing people to dispatch them to perform tasks that would otherwise consume great quantities of time (for example, doing market research, making a class project, etc). These qualities make them appealing for electronic marketplace applications [4].

However, mobile agent technology is not without its share of development issues. One of the most daunting problems - and one which is holding back widespread adoption in electronic commerce [2] - is the issue of trust. Because mobile agents are autonomous entities that work without supervision, they must be fully secure from attack and subversion. People are not likely to provide an agent with purchasing authority if they cannot be guaranteed that the authority will not be abused or compromised. Agent technology currently has several areas of security to shore up before they can enter mainstream usage [1][2]. Some of these areas will be explored in this work.

In order to explore the security issues related to agent technology, it is often helpful to view agents in the context of an application. In the present setting, we look at applying agent based technology to the performance of network vulnerability analysis. In short, this is the process of scouring a network and examining each host to see if there are vulnerabilities that could allow unauthorized access. As will be shown, this is an application that is particularly stringent on the security of the mobile agents that will be used as the underlying mechanism. This must be so. Vulnerability scanning and analysis is a very privileged operation that, if compromised, could open an entire network to hazard or damage. Like electronic commerce, it must therefore be very secure.

In the next section, definitions of some key concepts are provided. This is followed by a look at some of the pivotal security services that will be necessary for an agent based vulnerability scanning system. A description of two agent based systems for vulnerability scanning is then provided as a background for the section following in which various threat domains are introduced. A selection of techniques to combat these threats is provided within that section along with their applicability to agent based vulnerability analysis. This is followed by a summary and conclusions.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

2 Definitions

2.1 Agent Paradigm

In the seminal paper by General Magic [5], the mobile agent paradigm is described in familiar terms and concepts. A *place* is a location on a computer network that offers a service to a mobile agent. For the purposes of this work, a place is defined less abstractly as an *execution environment* which provides all of the necessary resources for an agent to execute properly. An *agent* is an independent entity that is to perform some work or task for the *agent originator*. It is composed of code, data, and state that will execute in a proper execution environment, or place. In this work, the task to be performed is to collect vulnerability information about the host on which each execution environment exists. An agent can *trave*l from one place to another in the performance of its duties. As well, when agents occupy the same place at the same time, the can hold *meetings*. They communicate through a common agent communication language.

In moving from a networked world dominated by a client server model to one dominated by fully *autonomous mobile agents*, there will be a multitude of intermediate stepping stones. One of these larger stepping stones is the concept of *static agents*. In a static agent paradigm, an "agent" travels to some foreign place and performs computations or tasks on behalf of the agent originator. The computation or work is performed at the execution environment and only the results are sent back to the originator. This has obvious bandwidth advantages over the client server model. However, the agent does not travel any further than the step taken to get to the target execution environment and generally has no ability to communicate with other "agents" existing in the same place. So, while a static agent shares some of the characteristics of the mobile agent paradigm (autonomy of computation, limited travel), it is not a fully qualified mobile agent.

2.2 Automated Vulnerability Scanning

2.2.1 Introduction

Automated Vulnerability Scanning (AVS) is a process of searching for system and network vulnerabilities that could allow malicious users or entities to obtain access or privilege that they would not normally be allowed to have. In the general AVS process, a database of tests is applied against each of the hosts on a network. Each test in the database is a script or a code that attempts to determine if a particular vulnerability is present on a given host or network device either by attempting to exploit the vulnerability itself or by searching for clues that the vulnerability exists (for example, the presence of certain files, modifications to certain files, etc.).

2.2.2 Evolution of Vulnerability Scanning

One of the first and best known AVS was SATAN (Security Administrator's Tool for Analyzing Networks)[6]. Since SATAN's debut in 1995, many other freeware and commercial AVS have

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

evolved but they all share a common methodology. As new vulnerabilities are discovered, test modules are written and incorporated into vulnerability databases which are then disseminated to the user community at varying rates (much like the current virus scanning model). Automated vulnerability scanning (also referred to simply as "scanning") is one of the most active and broadband methods for locating network vulnerabilities.

Initial vulnerability scanners were monolithic in that vulnerability scanning was performed using a single host to scan the entire network (Figure 1). This worked well where the networks were reasonably small, the bandwidth required for the vulnerability scanning was available (that is, unrestricted by network bottlenecks) and the privileges required for the scanning operation could be granted across multiple domains.

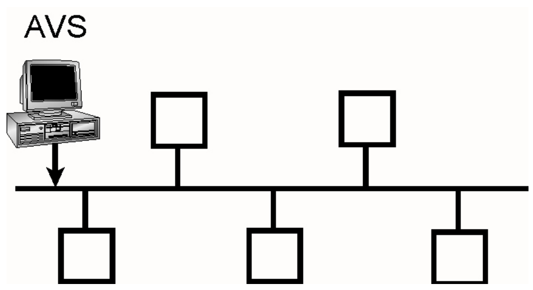


Figure 1
Single Host Network Vulnerability Scanner

As networks grew in complexity and size (both physically and logically), decentralized scanning tools were developed. For example, CyberCop Distributed 2.0 (now defunct), VigilantE SecureScan NX, Webtrend's Security Analyser, and Nessus provide a *console host* or *director* which can be used to coordinate the efforts of a number of distributed scanning hosts (Figure 2). Reports are collated by the distributed scanning hosts and forwarded to the director via some secure means.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

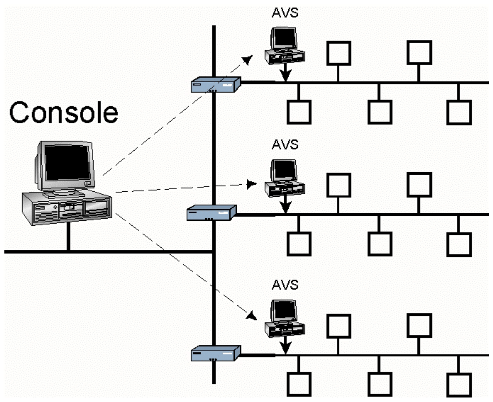


Figure 2
Distributed Network Based Vulnerability Scanner

The AVS tools thus far discussed are collectively referred to as "network" based scanners because they examine each host on the network from some remote location. As a result, they suffer from a limited ability to "see" beyond the network interface of any particular host. They must generally put more effort in detecting or inferring the presence of certain vulnerabilities than would be necessary if they had direct access to the inner workings of the host being scanned.[1]

To address this issue, a number of "host-based" vulnerability tools have been developed. This evolution essentially moves the AVS tools toward that of a static agent model as discussed earlier. With host-based AVS tools, a software agent, process, or script is distributed by a central authority to each host on the network (hereafter referred to simply as agent) where it remains indefinitely. The agent can be periodically updated by the central authority. It performs vulnerability checks at some predefined rate as a privileged user on the individual host (that is, from within the computer) and then reports results to the central authority (see Figure 3).

[1]Some scanning tools, such as CyberCop Scanner, require that the scanning tool have Administrator access and can use this access to “cross” the network interface boundary by inspecting registry keys etc.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

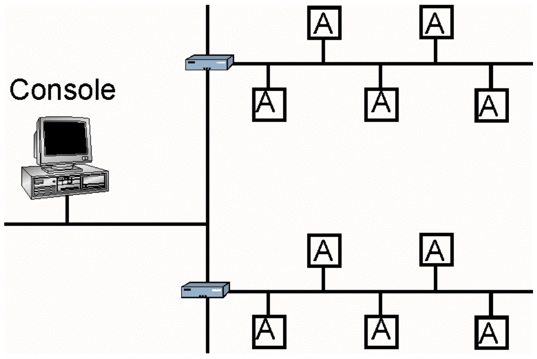


Figure 3
Host Based Vulnerability Scanner (where A represents an agent per host)

Host based vulnerability scanning agents are generally required to run with *full system access* and can perform security checks that are difficult or impossible for network based scanners to do. These include policy checking, detection of illegal software (network sniffers, password grinders, passive Trojans), and detection of illegal hardware such as modems. They can also use various techniques to actively search for evidence of malicious activity (for example, detecting the presence of altered binaries) and, in some cases, lock out offending users or processes[7]. Examples of host-based scanners include Harris Stat Neutralizer, TIGER, and Sun Microsystem's Bruce.

3 Evolution Toward Mobility

The next evolutionary step is to construct a host-based scanning agent that has mobility and that can interact within an agent based environment and that can take advantage of all of the opportunities that mobile agent technology makes available. As we shall see, there are some very difficult security related problems to overcome (in practical ways) before agents could be trusted with such a critical function as vulnerability scanning.

3 Important Security Requirements for Vulnerability Scanning

Depending on the application, various aspects of agent security may take on more or less importance. In a vulnerability scanning application, there are two services of fundamental significance. These are:

a.) Confidentiality of the collected data. It is paramount that no other agent or host be allowed to observe the data collected by a mobile vulnerability scanning agent. This would be akin to providing potentially compromising data to an attacker for free if that attacker is able to observe the agent as it passes.

b.) Integrity of both the generated reports and agent code. The integrity of the reports is essential to ensure that the data requested is the data received and that vulnerabilities are not being masked by a malicious entity. As well, the integrity of the agent code is

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

paramount. Because the agent is mobile, a malicious change to the agent code on one host could affect many other hosts on the agent's itinerary.

Strong arguments could also be made for the confidentiality of the agent source code as well. Vulnerability scanning agents will typically require greater access at higher levels of privilege than the average mobile agent and it would be advantageous to hide this from casual viewing. However, the functions they perform are not necessarily proprietary. In fact, algorithms for testing for the presence of a vulnerability are often published with vulnerability advisories. As will be seen, protecting the confidentiality of the agent code is typically a difficult problem.

1 Architectures for Agent Based Vulnerability Analysis

There are two works that shall be referred to at various points throughout the rest of this document. These works involve agent based systems for vulnerability scanning. Both will described briefly in the following paragraphs.

1.1 **SENSS Bruce**

The first work, by Crocker [8], involves integrity and confidentiality extensions of a commercially available, *static* agent based system for vulnerability scanning known as Sun Enterprise Network Service (SENSS) Bruce.

The Bruce tool is an example of a host based vulnerability scanning tool. Here, "host-based" is synonymous with "static agent" since there is executable code running on behalf of the tool on each of the individual hosts in the network. As we shall see, the executable code is not truly "mobile" since it is pushed to each host on the network in a direct fashion and, once on a host, does not autonomously move. Figure 4 shows how the system operates.

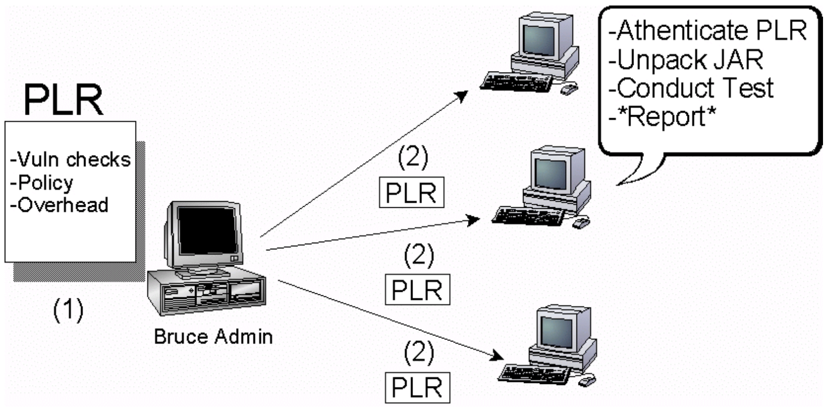


Figure 4
Sun Enterprise Network Service (SENSS) Bruce Tool

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

As described in [8], a Bruce Administration host creates a "pollet launch request" or PLR (1). A PLR consists of pollet "packages" containing executable code to perform one or more vulnerability checks. These checks are written in any number of languages from C to Java. Because the network to be checked is not homogeneous, the PLR also contains policy specifications concerning which packages are to be run on which types of host operating system. The PLR also contains overhead pertaining to secure transmission between hosts.

The PLR is passed to a process resident on each host of the network (2). After checking the integrity and authenticity of the PLR, the policy specifications are checked by the process and the appropriate executables are unpacked and run. The data that results from the vulnerability check is then stored locally on the computer. This data is collected at some later time by the Bruce Administration host. In the original distribution of the Bruce tool, the collected data remained in clear-text form on the hard disk of each host. At collection time, the data was forwarded to the Bruce Administrator in clear-text as well. The work by [8] involved overlaying a public key infrastructure and making use of it to protect the integrity and confidentiality of these reports.

Further levels of distribution can be achieved with Bruce by building a tree network. In this case, the Bruce Administration host would send the PLR to a number of "level 1" hosts which in turn would propagate the PLR to each of their children in "level 2" and so on. When it is time to collect the vulnerability data, each level collects reports from the level below it and forwards these upwards upon request.

4.2 Mobile Vulnerability Scanning Agents

The second work by [9] was the only documented attempt found thus far to implement a vulnerability scanning system based on fully mobile agents. The system uses Java and is composed of several components interconnected in the fashion shown in Figure 5 below.

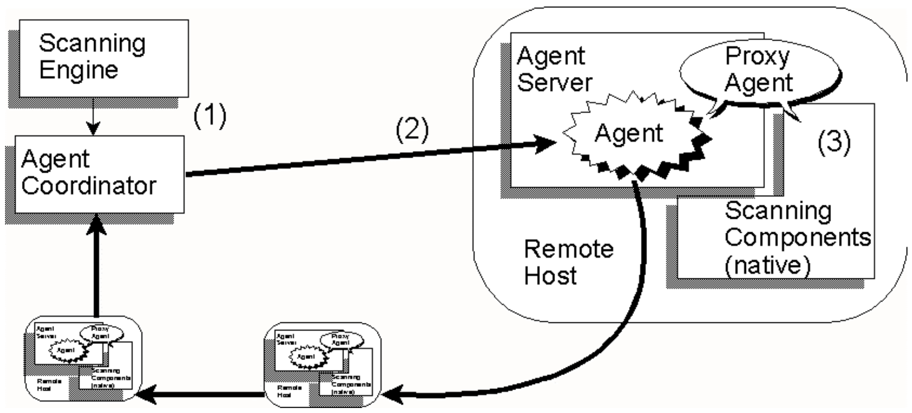


Figure 5
Humphries, Pooch, and Carver Mobile Agent Based Vulnerability Scanner

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

An agent coordinator communicates with a vulnerability scanning engine (1). This engine through the agent coordinator, generates agents as required to fulfill a vulnerability scanning task. The agent coordinator is responsible for maintaining all configuration information about all of the components in the system and for tracking agents throughout their life.

At creation time, agents are dispatched in a secure, authenticated fashion to the first host on the itinerary (2) where the host execution environment, called the “Mobile Agent Server” in [9] takes control. Because the agent is written in Java, it does not have privileged access to system resources that are required to perform the tasks necessary of a vulnerability scanning system. As a result, the Agent Server provides visiting agents access to locally resident vulnerability scanning components (written in native code) via trusted local proxy agents (3).

Agents collect and securely store payload data from interactions with the trusted local proxy agents and then are moved between hosts on the itinerary by each Agent Server (4). When the agent returns to the Agent Coordinator, the payload for each host is provided to the vulnerability scanning engine for analysis.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

2 Threat Domains

The following represents a generally accepted taxonomy or categorization of attacks [1][10][11] that mobile agents can be involved in:

a.) An agent attacks a host when it arrives on that host;
b.) A host attacks an agent when the agent arrives;
c.) An agent is attacked by the environment (usually during transit); and
d.) An agent attacks another agent

In this work, much of the focus will be given to the first two types of attacks since these seem to encompass the majority of the current research. The third category will be touched upon briefly to indicate potential solutions that are applicable from the first two categories. Techniques to address the fourth category will not be elaborated upon.

The techniques identified in this work can be used to either *prevent* or to *detect* certain attacks and, where possible, a distinction shall be made. Tan ([12]) has noted that, in general, preventive techniques are most reliable but tend to be complicated and expensive to deploy because they assume that no entity is trusted. On the other hand, detection techniques are generally more easily deployed and can be used to interactively modify the trust levels of certain hosts when trouble does arise.

In the following sections, each of the categories will be examined separately in the context of using mobile agents for network vulnerability analysis. A brief description of the problem domain and potential attack scenarios will be given followed by a description of a number of techniques that exist (or are currently under study) to mitigate potential attacks. Finally, a discussion of the utility of each identified technique for use in a vulnerability analysis context will be provided.

2.1 Agent attacks Host

2.1.1 Description

The central problem in this category is for the host to determine if the mobile agent can be *trusted* to execute in its environment without doing harm. This is a very well researched problem domain [12][13] because it is a current issue with the use of internet browsers. A client browser can be configured to download and execute "active" code (ActiveX, Java, etc) on the client computer and some assurances that this code is not malicious have been demanded by application consumers.

2.1.2 Attacks

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Once an agent has been executed on a host, it can perform a number of malicious activities. These can include masquerading as another agent to obtain data or perform some action, deny service to other agents or the system by consuming excessive resource, or making unauthorized access to system data or resources with the possibility of doing permanent damage [10].

In the context of vulnerability scanning, masquerade and unauthorized access types of attack are particularly troublesome. If an attacker can masquerade vulnerability scanning agents, then he/she can not only obtain critical information about the vulnerabilities of the network, but he/she can do it with the *direct* cooperation of each host on the network (and very likely without detection) [14]. And, as will be discussed shortly, a vulnerability auditing agent requires considerably fine-grained access to very critical system resources in order to do its job - such access can be extremely damaging in the wrong hands. Finally, while denial of service attacks can be troublesome, they tend to be very visible and unmask the attacker or compromised hosts quickly. On a closed network, this problem can ordinary be dealt with rather quickly through policy or other processes.

2.1.3 **Sand-boxing**

2.1.3.1 **Description of technique**

Sand-boxing ([3],[15],[16],[10][17],[18]), in a general sense, is the act of confining a visiting mobile agent to a certain execution space and allowing or denying that agent access to specific, potentially harmful, resources [3]. Examples of potentially harmful resources include access to the file system, network sockets, or operating system resources. Sand-boxing is a *preventative* technique that seeks to halt malicious agents a the operating system interface.

The JAVA interpreted language appears to have become an almost defacto platform for mobile agent execution environments. Indeed, Altmann et. al. [19] surveyed over thirty JAVA based agent development platforms. This is probably due to a number of factors including it's platform independence, wide-spread application, and it's multifaceted sand-box security mechanism. The work in [8] and [9] are both based on Java. For the purposes of discussing sand-boxing, a description of the Java JDK 1.X sand-box mechanism will now be given

The JDK 1.0.x sand-box comprises a number of cooperating system components that work together to inspect, review, and restrict the execution of remotely received applets [17],[20]. Before execution, the code comprising the agent (JAVA byte-code) is put through a static inspection to ensure strong type safety and to ensure that the code adheres to all of the rules of the language. The inspection mechanism is paranoid; it assumes that the code was not written by a JAVA compiler and that it is meant to crash or penetrate the system's security features. This static inspection relieves the run-time system from having to keep an eye on certain types of interactions.

Agent code is provided with a separate, distinct name-space through class-loading and this is the

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

mechanism that controls what other portions of the runtime environment that the code can access and modify (for example, other memory, processes, etc). This code is also prevented from invoking methods that are part of the system's class loader (to prevent the code from defining its own sand-box). Finally, a *security manager* mediates all accesses to system resources. It performs run-time verification of dangerous methods (file I/O, network access, definition of new Class Loader, etc). In essence, the JDK 1.0.x sand-box could be likened to a safe code interpreter and reference monitor.

2.1.3.2 **Application to Vulnerability Scanning**

The use of a sand-box in the execution environment of a mobile vulnerability analysis agent is problematic. By its very nature, a vulnerability analysis agent will attempt to make access to often restricted or very sensitive system resources and, depending on the native operating system, these accesses may have to be performed with maximal system privilege. If the agent is restricted to a tight name-space and nominal privilege, then it may not be able to perform its duties.

The problem would thus seem to be one of managing privilege and access and attempting to determine what constitutes "normal" or "tolerable" behavior for a vulnerability analysis agent. An interesting approach in this direction is presented in [21]. Here, a sand-box is "custom built" around an un-trusted third party software (sendmail) by performing a privilege and resource analysis on it, constructing a policy definition, and then developing a sand-box to confine software and established boundaries. Such an approach would be useful in developing a sand-box for vulnerability analysis agents if the privilege and resource requirements could be bounded in such a way as to allow flexibility for a constantly changing vulnerability environment.

The use of platform independent languages such as JAVA for developing mobile vulnerability agents poses another concern for implementing sand-box architectures. Because, the language is platform independent (and therefore oblivious of system architecture), there is no code base to assist the agent in conducting the vulnerability analysis [22]. In both [8] and [9], this limitation is bypassed by allowing the vulnerability analysis agent to effectively "step out of the sand-box" by invoking native methods (which exist on the host platform or are delivered along with a scan request). When native code is executed, the Java sand-box no longer exercises any control over what resources it accesses.

2.1.4 Authentication of the Agent Through *Code Signing*

2.1.4.1 **Description of technique**

Code Signing is the process whereby the producer of a mobile agent constructs a digital signature on that agent for *authentication* (and integrity) purposes. Execution environments can *authenticate* the source of the code based on this signature and decide whether to execute the code or not. A signature does not indicate if the code is malicious - it just verifies that is was

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

created by the owner of a certain private key [3]. Figure 6, which was inspired by descriptions in [23], illustrates code signing.

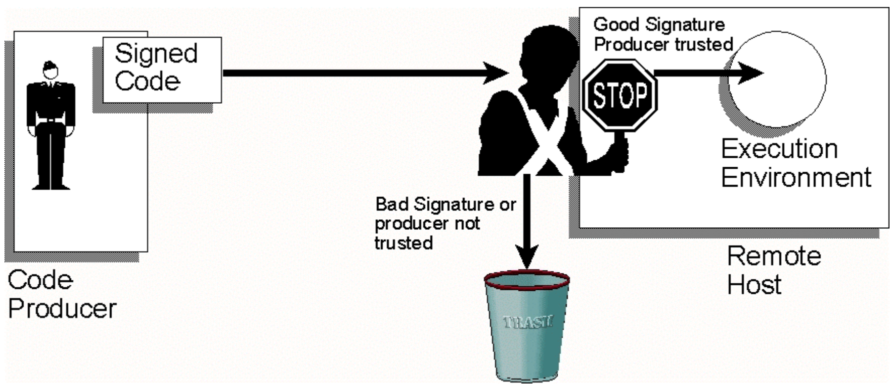

Signed Code
Code Producer
Good Signature Producer trusted
STOP
Execution Environment
Bad Signature or producer not trusted
TRASH
Remote Host

Figure 6
Agent Authentication Through Code Signing

Authentication is labeled as a preventative technique here because, if the execution environment determines that it cannot "trust" the agent that just arrived, then it can choose to either deny the agent altogether (or let the agent run with severely restricted privileges depending on the complexity of the system trust model). Digital signatures are the most common methodology for authentication [18] and these are derived from techniques in public key cryptography.

A digital signature of a message can be formed as shown in Figure 7. Assume Alice wants to send Bob a piece of executable code and Bob would like to ensure that the code came from Alice. Alice passes the code through a one-way hash function (1) (often referred to as a "message digest") which represents a fingerprint of the original code [24][25]. The hashed value can be used to guarantee the integrity of a message since the hash value is unique; making even the smallest change to the message will result in a different hash. Alice then encrypts the hash using a private key and some asymmetric encryption algorithm (2).

Bob receives both the original code and the hash of the code. He passes the code through the same hash function. He uses Alice's public key to decrypt the signature to obtain the hash computed by Alice (3). If this hash is the same as the one Bob computed, then Bob knows that the code was sent by Alice and that it was not modified in transit.

This operation represents the essence of signed code [10]. The technique of code signing is also often referred to as "digital shrinkwrap" and is the model for MicroSoft Authenticode Active X controls [15][26].

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

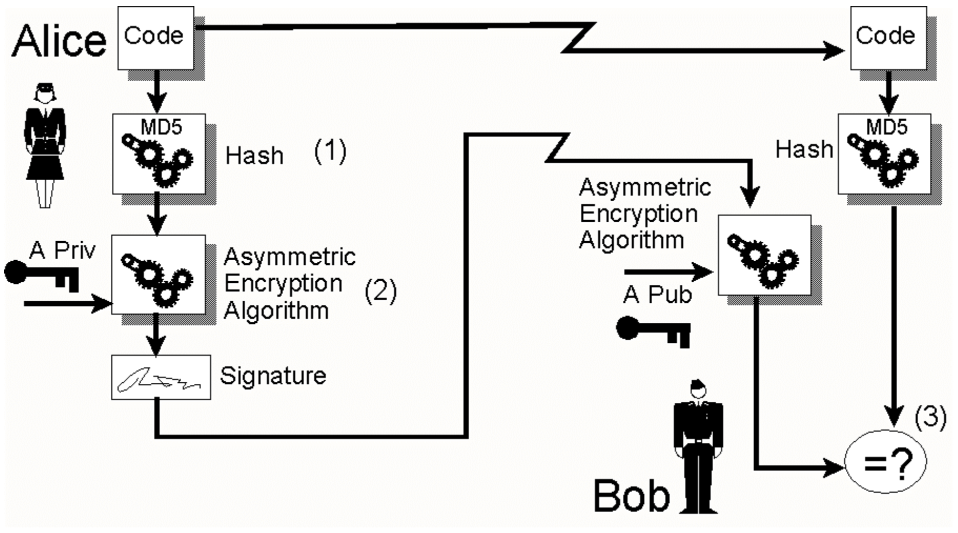

Alice
Code
MD5
Hash
(1)
A Priv
Asymmetric Encryption Algorithm
(2)
Signature
Code
MD5
Hash
Asymmetric Encryption Algorithm
A Pub
(3)
=?
Bob

Figure 7
Formation of a Digital Signature

Digitally signing code, as described above, makes strong assumptions about the ability to secure private and public keys. For instance, we may ask how Bob gets Alice's public key. The worst possible situation would be for Alice to send the public key with the code. In that case, Eve - who is listening to the channel and is in a position to perform a "man in the middle" attack - can simply modify the code, re-compute the hash, and then re-sign it with her own private key. Bob will faithfully use Eve's public key (provided as part of the modified message) in the belief that it belongs to Alice unless he has some way of verifying that the key being used belongs to Alice.

A potential solution to securing and "trusting" public keys is to incorporate some form of public key infrastructure (PKI) [27][24]. In a PKI, "certificates separate the signing and lookup functions by allowing a certificate authority to bind a name to a key through a digital signature"[28]. In the context of Alice, Bob, and Eve, the authenticity of Alice's public key is certified by a third party who signs Alice's public key (and other information describing Alice). This certified document is called a digital *certificate*. The third party (or certificate authority) places Alice's certificate in one or more publicly accessible locations. The certificate authority also provides its public key in that repository. Eve cannot modify Alice's certificate without knowing the trusted authority's private key - which is closely guarded.

2.1.4.2 **Application to Vulnerability Scanning**

Vulnerability agent authentication through code signing is a technique that can be implemented in a vulnerability analysis system now (subject to the availability of a minimal PKI) and represents one of the best methods for preventing attacks by agents on hosts. This does not necessarily have to be a full implementation of X.509. Indeed, simplified infrastructures such as

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

SPKI/SDSI [29] or perhaps customized elements of X.509 may be sufficient to meet the needs of this relatively specific application.

The major disadvantage of signed code from a vulnerability analysis perspective is that the trust model for code signing is absolute - it assumes that it is possible to “distinguish trustworthy from untrustworthy [authors] and that trustworthy authors are in-corruptible.”[15] This presents two major points of compromise. First, if the agent originator is malicious (we can imagine a disgruntled vulnerability test script writer who has turned to the “dark side”), then code signing will neither detect his/her activities nor prevent them; total trust is held by such persons.

Second, if a host execution environment is compromised, then the secret signing key held by that host could also be compromised. Code signed by the compromised host will still appear to come from trusted entities. While the main body of agent code is protected by the agent originator’s signature, a hacker on the compromised host can still modify agent state information prior to forwarding it [30] (signatures on this information are only valid between hosts because state changes as the agent propagates). This may impact how the agent ultimately behaves on subsequent hosts. Also, the hacker can falsify and sign reports (for that host) using the compromised signing key. Additional methodologies to detect or mitigate tampering or malicious behavior should be employed in concert with code signing. Some of these methodologies will be discussed in the section on protecting the agent from the host.

The Bruce application described in [8] (version 1.0EA4) implements code signing for the authentication of pollets and pollet launch requests. Instead of having a central repository of certificates, each host is provided (out of band during installation) a copy of a “Master” certificate which is used to authenticate several lower level certificates which are then used to authenticate both pollets and pollet launch requests as they arrive at a host. The work in [8] seeks to merge the Bruce tool with the Entrust PKI in order to address confidentiality and integrity issues with the generation, storage, and transport of the vulnerability reports[2].

The work in [9] implements support for code signing. In this work, the code for each mobile agent is signed by an *agent coordinator* before the agent is launched. The code can be authenticated by any receiving host using the agent coordinator public key. In addition, before an agent migrates from one host to the next, the source host will re-sign the code. The destination host can authenticate that the agent came from the sending host using the sending host public key to verify the signature and check the integrity of the code. Unfortunately, details of key management are not provided in the published work so it is not entirely clear how public keys are obtained by the execution environments in trusted ways.

[2]There are ulterior motives for integrating the Entrust PKI with the Bruce tool. The reader is referred to [Crocker] for these.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

2.1.5 **Proof Carrying Code**

2.1.5.1 **Description of Technique**

Proof carrying code (PCC) is a technique aimed at *preventing* malicious activity by a visiting mobile agent on a host execution environment. The key idea of PCC, as proposed by Necula and Lee [31][32] is that an agent *producer* will build and attach a proof to a mobile agent. This proof guarantees that the agent conforms to the *safety policy* of a code *consumer* (the host execution environment). When the mobile agent resides on a remote host, the attached proof can be validated by the host and, if it passes, the agent can then be executed safely without any further checking.

In this technique, the bulk of the work of ensuring that the code behaves according to established policies has essentially been relocated to the agent originator (unlike a sandbox or reference monitor where the work is done locally). In [32], the following advantages of relocation in this manner are identified as follows:

a.) The proof checking code (the portion existing on the host execution environment) can be small, automatic, and relatively simple which makes it easier to verify and manage from a security point of view. As indicated in [33], this could be several thousand lines of code versus several tens of thousands of lines;

b.) Because the proof checking code is small, the proof checking process can be fast;

c.) The agent code runs at native speeds since, once the proofs are validated, no further monitoring is required [34].

c.) No trust relationships with the agent code producer or proof producer is required.

The PCC process, as described in [31], is outlined in Figure 8. The Figure shows the transactions that take place between an agent originator and a remote execution environment during a PCC exchange.

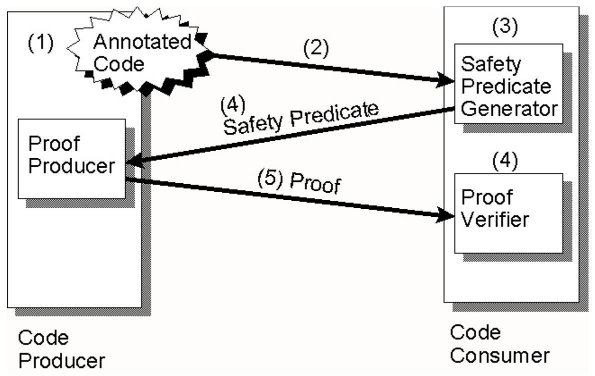


Figure 8
The Proof Carrying Code (PCC) Process

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

In the first step, the agent originator produces the agent code (1). The agent originator attaches certain annotations to the code in order to assist the generation of a safety predicate (which is described later). The agent producer then sends the code to the remote execution environment (2).

The remote host execution environment has one or more pre-defined safety policies. The definition of a safety policy is the responsibility of the host execution environment and it specifies the conditions under which it considers the execution of the agent code to be safe. As described in [31], safety policy is composed of three elements:

1.) A verification condition generator which is a device to compute a safety predicate in first-order logic based on the code to be proved;
2.) A set of axioms and inferences that can be used to prove the safety predicate; and
3.) Pre and post-conditions for the agent code. These are conditions that must be met by the agent code before and after execution.

For each instruction in the agent code that may potentially violate the safety policy, a verification condition generator produces a *verification condition* that describes the circumstances under which the execution is safe. The collection of verification conditions composes the *safety predicate* of the agent executing on the remote host (3).

The execution environment then sends the safety predicate to a *proof producer* (4). In Figure 8, this is shown as being the agent originator but Necula and Lee are careful to indicate that this need not be so. It could, in fact, be any other machine on the network. The proof producer is a theorem prover for first order logic that attempts to build a proof of the safety predicate using the axioms and inference rules that form the safety policy of the remote host.

If the code producer is successful at building a proof of the safety predicate, then the encoding of the proof is sent back to the host execution environment (5). Note that a safety predicate is a specific encoding of the safety policy of the remote execution environment for a particular agent. The system can be simplified somewhat if the agent producer knows the safety policy of the remote host execution environment and can produce the safety predicate locally. If every host execution environment has the same policy, then the agent producer can generate the safety predicate, send it to the proof producer (if it is not co-located) and then attach the proof directly to the agent before the agent is sent to the remote host(s).

The final step in a PCC system is validation of the proof by the host execution environment (6). The validation step ensures that the proof does, in fact, prove the safety predicate that was originally generated (that is, to ensure that the proof is not for some other safety predicate) as well as ensuring that all of the inferences and axioms within the proof are valid as specified by the safety policy. If all of this activity results in a successful proof of the agent code, then the agent is allowed to execute without further interruption or monitoring.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

2.1.5.2 Application to Vulnerability Scanning

PCC does not yet appear to be mature enough to be used in a general purpose agent based application. Some hurdles to implementation include the necessity of developing a standard formalism for establishing security policy as well as a fully automated means for the generation of proofs [10].

On a typical network to be audited for vulnerabilities, there is usually a large range of operating systems and hardware architectures. At first, it may seem that this issue would pose a problem for PCC since it is hardware dependent. However, hardware dependence gives PCC an advantage over hardware independent programming environments (for example, Java). PCC, through the execution environment's policy, implements fine grained control of the agent code. The author of [10] indicates that it might even be a "reasonable alternative to software based fault isolation." As previously discussed, vulnerability analysis agents need privileged access to system resources that cannot be accommodated by hardware independent programming environments such as Java. And, while *access* to native methods could be controlled in such environments, once they are invoked all control is lost until they return. Therefore, trust in the native methods would necessarily be absolute and this trust opens a vector of attack. With PCC, trust in the *entire code* is developed as a result of the correctness and validity of the corresponding proof. There is no opportunity to target and attack portions of the code that may not be subject to monitoring or correctness since none exists.

2.2 **Host attacks Agent**

2.2.1 **Description**

Protecting a host execution environment from a malicious agent is well studied and the problem domain could arguably be described as solved. The opposite situation is, unfortunately, not as well in hand. Protecting mobile agents from malicious hosts appears to be a very hard problem indeed [18][35][2]. The difficulty arises for two main reasons. The first is that if an agent ends up residing on a compromised (that is, a malicious host), then that host can make use of all of its local resources (including time) to attempt to compromise the agent [36].

The second problem is that the agent based paradigm advocates the autonomy of mobile agents. This means that any techniques that a mobile agent might use to protect itself from potentially malicious execution environments must be built into the agent and ported around with it as it moves from host to host. This can create a range of other problems including (but certainly not limited to) bandwidth limitations (if the agent becomes too large) as well as the examination and/or tampering of the agent protection mechanisms when residing on a malicious host. Clearly, the problem is lopsided in favor of the execution environment.
There are currently four research areas that attempt to deal with protecting an agent from its execution environment. These are identified in [2] and summarized here. The first involves extending the trusted computing base [18] so that all hosts are trusted and all communications are

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

secured. The second approach is a derivative of the first where agents can migrate to hosts that are trusted or that may not be trusted initially but have a good reputation. Clearly, these approaches are inappropriate for a system of vulnerability analysis agents since the model inherently assumes that any or all hosts could be compromised. The third approach focuses on detecting malicious acts against the agent while the fourth approach seeks to secure the agent from prying eyes while existing on a remote execution environment. These last two approaches will be considered here.

2.2.2 **Attacks**

A categorization of attacks that can be perpetrated by an execution environment against an agent are presented in [37] and listed below. An execution environment can:

1.) Spy out agent control code;
2.) Spy out agent data;
3.) Spy out agent control flow;
4.) Manipulate agent code;
5.) Manipulate agent data;
6.) Manipulate agent control flow;
7.) Incorrectly execute agent code;
8.) Masquerading itself as another host to the agent;
9.) Deny execution of the agent;
10.) Spy out interaction with other agents;
11.) Manipulate interaction with other agents; and
12.) Returning wrong results of system calls issued by the agent.

In the context of a mobile vulnerability scanning agent, eavesdropping and alteration of data by the host (items 2, 5) can be extremely bad and may even *assist* a malicious entity in compromising other hosts. For example, eavesdropping and alteration of agent data could allow a malicious host to falsify the data collected by an agent thereby “hiding” any of the vulnerabilities that may have allowed the host to be compromised in the first place.

Since the vulnerability scanning agent is mobile, it may also be possible for a malicious host to examine and modify the vulnerability data collected on *other* hosts. The vulnerability analysis agent will be performing reconnaissance against the network *on behalf of* the malicious host. This reduces the risk of being detected by an intrusion detection system since vulnerability data is being collected passively.

Finally, if the compromised host can alter the reports or partial results carried by a vulnerability scanning agent as it passes through, then it could effectively hide vulnerabilities from the detection system with the intention of using them to leverage further network access. It is therefore very important that the integrity and confidentiality of data collected by vulnerability analysis agents be preserved.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Eavesdropping and alteration attacks against the vulnerability analysis agent code (items 1, 3, 4, 6, 7) can also be extremely dangerous. If a mechanism is not in place to guarantee that altered code cannot be passed on to other hosts, then the attacker will have achieved the "holy grail" - getting his/her code to run on other machines. This problem can be exacerbated by the fact that the vulnerability analysis agent may be required to run with full system access in order to perform its work; the attacker does not have to leverage further access on remote hosts once he/she "owns" a privileged mobile agent. Clearly, alteration attacks must either be prevented from occurring or altered agents must be prevented from propagating.

It is not clear how useful it is to try to prevent a compromised execution environment from reading or modifying an agent code *while the agent is resident on that host* (as long as modified agent code could not be propagated). This is due to the fact that, since the host environment is already compromised, it is not necessary for the host to try to leverage the agent to gain further access. The vulnerability checks themselves do not generally constitute proprietary methods and hence need no protections. However, in preventing an agent code from being intelligently read and interpreted, some protection against alteration is provided since it is difficult to intelligently modify an entity that is not clearly understood.

2.2.3 **Code Hiding**

In this section, three methods of "code hiding" are discussed. These are encrypted functions, code obfuscation, and time limited black boxes. All are techniques that share a common goal: to prevent the a potentially hostile execution environment from determining what the agent is doing on behalf of a user.

As mentioned previously, an important result of any hiding effort is to prevent an attacking host from intelligently modifying agent code (both in execution and prior to dispatch to the next host). We saw that this type of integrity could be secured using digital signatures but that verification of these signatures required some form of infrastructure for trusted distribution of public keys. Code hiding would not require such an infrastructure.

Depending on the system that is implemented, a vulnerability scanning agent may be required to carry a secret (for example, a private signing key) in order to perform its work. Since the agent executes the cryptographic primitive in plaintext, it is possible for the hostile execution environment to "steal" this secret. Code hiding could help the agent keep the secret confidential.

Each of the hiding techniques will be explained in the following paragraphs and their applicability (as a family) to the vulnerability analysis task will be discussed. All of these techniques are *prevention* methods.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

2.2.3.1 Computing With Encrypted Functions

2.2.3.1.1 Description of Technique

In this technique, the goal is to encrypt a target function such that the transformed function can again be implemented as a workable program [35]. This program consists of plaintext instructions that a computer can understand but which does not reveal the original function.

Consider the following simple example posed by [35] and paraphrased here. Assume that the function Alice wants to evaluate on Bob's computer using Bob's input x is a linear map M. She does not want Bob to know what the map function is, so she picks a random, invertible matrix S and computes B:=SM. She composes a program to compute P(B) on some input x and then sends the program to Bob. Bob executes P(B) on x and essentially obtains y:=Bx which he sends back to Alice. Alice then performs $S^{-1}y=S^{-1}(Bx)=S^{-1}(SMx)=Mx$. Alice obtains Mx without revealing to M to Bob. This process is shown below in Figure 9.

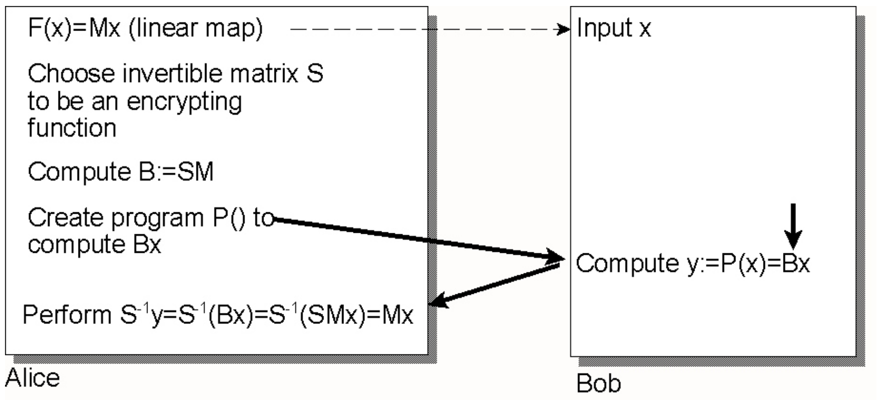

F(x)=Mx (linear map)
Choose invertible matrix S to be an encrypting function
Compute B:=SM
Create program P() to compute Bx
Perform S⁻¹y=S⁻¹(Bx)=S⁻¹(SMx)=Mx
Alice
Input x
Compute y:=P(x)=Bx
Bob

Figure 9
An Analogy for Computing With Encrypted Functions (from [35])

The work in [35] assumes, for the general case, that the function to be encrypted is rational and that the encrypting function (the function that would perform as the invertible matrix in the example above) is also rational and efficiently invertible. The security of the method is then based on the difficulty of decomposing the rational function that results from the product of the encrypting and target functions. However, they do identify that there is a challenge in finding encryption schemes for arbitrary functions and acknowledge that the problem remains fundamentally open.

An incremental work in [38] uses exponential functions as encrypting functions (instead of rational functions) in a proposal for a secure solution to creating undetachable signatures. The challenge is to allow a mobile agent to effectively produce a digital signature inside a remote host

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

without the host being able to determine the agent's secret or to reuse the signature routine for arbitrary documents (hence, undetachable). The undetachable signature scheme is based on the RSA cryptosystem and the server (agent originator) binds a Customer to a Transaction via a set of customer constraints to prevent the signature routines from being reused.

2.2.3.2 **Obfuscation or Code Mess-Up**

2.2.3.2.1 **Description of Technique**

The ultimate goal of code obfuscation is to construct an obscured version of a program, P, such that an examination of the observed code yields no useful information about what the code does. This is essentially the same goal as that of encrypted functions except that obfuscation makes no pretense of being provably secure. The obfuscation effort seeks to make the work involved in reverse engineering the code as difficult as possible and, ideally, as difficult as doing a black-box study of the code [36]. Such a study involves observing the input/output relationships of the obfuscated code over many executions and with different stimuli to attempt to infer program functionality.

A taxonomy of obfuscating transformations have been identified by [39]. This taxonomy is summarized below.

Layout Obfuscation: Obfuscation of information that is redundant to the execution of the code (for example, identifier names, comments, etc). This type of obfuscation is considered trivial and can be easily by a determined attacker.

Control Obfuscation: This involves attempts to obscure the control-flow of the source program and depends to a large extent on the existence of what the authors of [39] call opaque predicates. A predicate is opaque if its outcome is known at obfuscation time, but is difficult for a reverse engineer to deduce.

Consider the example shown in Figure 10 (reprinted with permission from [39]). We have predicates, P, that may always evaluate to TRUE, FALSE, or to one or the other. We denote these as PT, PF, andP? In Figure X, we have a section of code that can be broken down into two subsections, A and B. We can split the code by inserting a true predicate as shown to make it appear as though B is only executed sometimes. Alternatively, we can generate two different obfuscated versions of B and then use the predicate P? to select between them at runtime. Finally, we can introduce a "buggy" version of B and select PT such that it appears but is never selected to run.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

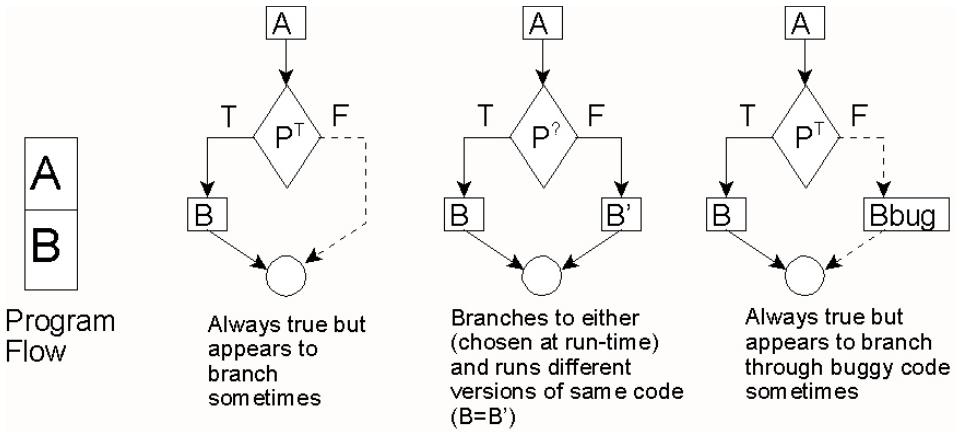
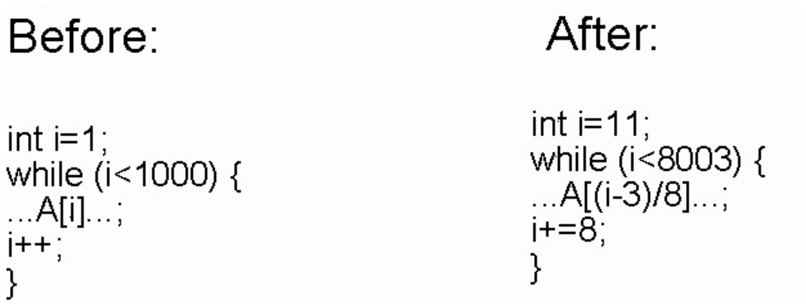

A
B
Program Flow
A
T
F
P^T
B
Always true but appears to branch sometimes
A
T
F
$P^?$
B
B'
Branches to either (chosen at run-time) and runs different versions of same code (B=B')
A
T
F
P^T
B
Bbug
Always true but appears to branch through buggy code sometimes

Figure 10 (Figure 4 of [39] with annotations)
The Use of Opaque Predicates to Obfuscate the Program Flow AB

Data Transformation: This involves transformations that obfuscate data structures or to obfuscate how data is interpreted. There are several categories of data obfuscation and these are summarized by [40] as follows:

1.) A *data storage* obfuscation might be to change a local variable to a global one.

2.) A *data encoding* obfuscation will affect how a stored item is interpreted. Consider an example where we replace the integer variable i with 8***i**+3 and then re-implement the code as shown in Figure 11 below (code snippet reprinted with permission from [40]). Clearly, the role of **i** in the revised code has been made somewhat more difficult to determine.



Before:

```
int i=1;
while (i<1000) {
...A[i]...;
i++;
}
```

After:

```
int i=11;
while (i<8003) {
...A[(i-3)/8]...;
i+=8;
}
```

Figure 11 (code snippet from [40])
Example of a Data Encoding Obfuscation

3.) A *data aggregation* obfuscation changes how data is grouped. An example would be changing a two-dimensional array to a one-dimensional array (or vice versa).

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

4.) A *data ordering obfuscation*, as its name implies, changes how data is ordered. For example, we could use a generic function F() to determine the position of the **i**th element in a list instead of indexing that element using the value **i**.

2.2.3.3 **Time Limited Black Box Security**

2.2.3.3.1 **Description of technique**

The concept of time-limited blackbox security, as outlined in [41] is very similar to that of code obfuscation; to create an executable agent which is immune to observation or (intelligent) manipulation attacks while existing on a potentially malicious host. However, the objective of pure code obfuscation is to hide the functions or algorithms (or the "intellectual property") within the agent for as long as possible (preferably indefinitely). On the other hand, time limited blackbox security is a *general* technique that seeks to match the length of time that the functions and data within the agent should be hidden with the length of time that the agent will be useful. In other words, the agent code or secret is confidential for a fixed period of time after which nothing of use can be gained by observing the code. The perceived benefit is a potential reduction in the cost (both in conversion time and code size) of hiding the agent secrets.

An agent is described by [41] as having the blackbox property if the code and data of an agent cannot be read or modified. A "conversion mechanism" (a specific instance of such a mechanism would be a code obfuscator) creates a blackbox agent from an agent specification (the agent code) and a set of initial parameters. An important property of this conversion mechanism indicated by the authors in [41] is that it is capable of producing *different* blackbox agents from the same initial agent specifications (source code). This prevents attacks in which many agent specifications are applied to the conversion mechanism to produce a library of blackbox agents that could then be used to characterize an unknown agent (assuming that the conversion mechanism is available to an attacker).

In order to be a candidate for the time limited black box approach, an agent must not be composed of intellectual property or code that cannot *ever* be revealed. This should be quite obvious. Additionally, there must be a reasonably accurate way of estimating the amount of time that is required for the agent code to remain hidden. This estimation is dependent on an evaluation of the skills and abilities of reverse engineers and on the strengths of the conversion mechanism. The authors of [41] acknowledge that this is a difficult to do with any accuracy.

2.2.3.3.2 **Application to Vulnerability Scanning**

Computing with encrypted functions does not yet appear to represent a technique mature enough to be employed as an element of a mobile agent based vulnerability analysis tool. However, it does represent the ultimate in terms of the ability of an agent to defeat observation or alteration attacks by an execution environment because of the purported security of the method.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

If a solution for general functions is found, its applicability in a VA sense will still be dependent on flexibility and ease of implementation. There will be many vulnerability checks performed each day by different vulnerability agents and their timeliness will often be crucial; the time between the public announcement of a vulnerability and the first probes for that vulnerability can often be measured in hours. Hence, unless the conversion of a program to its encrypted state is automatic and fast, the benefits of doing so may not outweigh the pressing need for speed.

Code obfuscation offers a practical and currently available mechanism for confidentiality (and to a large degree integrity) of vulnerability scanning agents. This is especially true if the code is written in Java since there appears to be a large number of commercial offerings for automated Java code obfuscation (for example [42][43][44][45]). However, it appears that the general techniques of code obfuscation could be applied to most classes of programming language and there are commercial ventures available to this end (eg. [46]). Thus code obfuscation offers itself as a more practical solution to hiding the nature of the agent code than does encrypted functions.

In both encrypted functions and code obfuscation (assuming that a general solution to encrypted functions is found), applicability towards vulnerability analysis will ultimately be dependent on flexibility and ease of implementation. There will be many vulnerability checks performed each day by different vulnerability agents and their timeliness will often be crucial; the time between the public announcement of a vulnerability and the first probes for that vulnerability can often be measured in hours. Hence, unless the conversion of a program to its encrypted or obfuscated state is automatic and fast, the benefits of doing so may not outweigh the pressing need for speed. Additionally, the cost of encryption or obfuscation in terms of additional code size must be reasonable since the agent will be mobile and will exist with other agents competing for bandwidth. If these constraints could be met, then encrypted functions and/or code obfuscation could prove to be useful elements of a vulnerability analysis process based on mobile agents.

Finally, elements of the general approach of time-limited blackbox security could be of particular use in a vulnerability scanning system. If code obfuscation is the primary means of ensuring confidentiality and integrity of the agents, then it would be useful to place an upper limit on agent lifetime. Although this would require a synchronized time-base, it could prevent replay attacks for code that had been reverse engineered.

2.2.4 Protecting Agent Data with Public Key Encryption and PRACS

2.2.4.1 **Description of Technique**

Partial Results Authentication Codes (PRACs) allow an agent to carry the signing keys necessary to vouch for the integrity of the data it collects. The agent may also carry the public key of the agent originator in order to protect the confidentiality of the results. Alternatively, this public key may be held by the execution environment on each host.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

In its simplest implementation, dubbed “Simple MAC-based PRACs” in [30], a set of secret keys is generated by the originator of the agent and two copies are made (see Figure 12). One copy is given to the agent and one is retained by the originator. There is exactly one key per host on the agent’s itinerary.

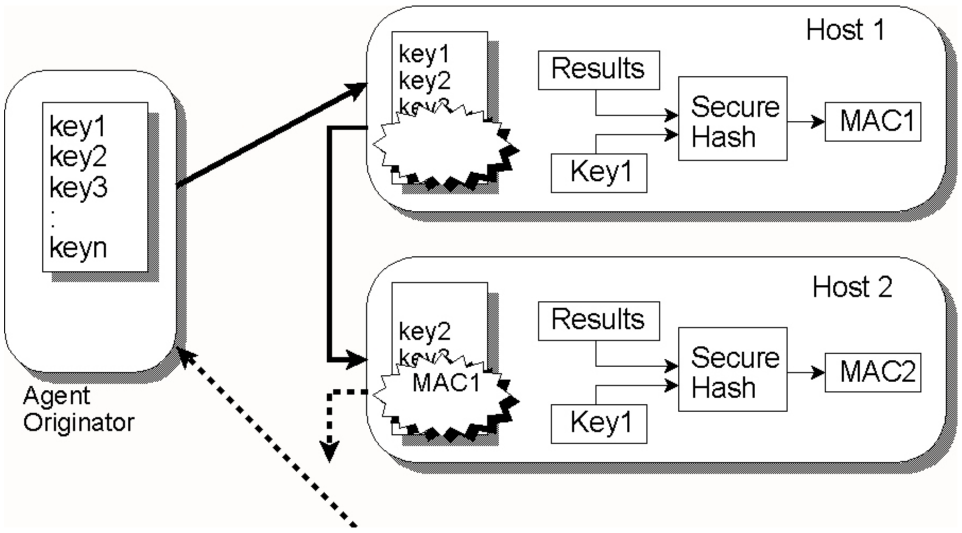


Figure 12
MAC Based PRACs

In the Figure, the agent is at Host 1. There, it gathers results and then computes a “message authentication code” (MAC1) on those results using the first key (key1). Computing a MAC is similar to computing a hash function except that it has two inputs - the original message and a secret key. It produces a smaller, fixed size output that is practically infeasible to reproduce without knowledge of the secret key [25]. The only entity that can reproduce MAC1 (and thereby verify the integrity of the original message) is the holder of the same secret key - the originator. Once the mobile agent has computed MAC1 on the results collected at Host 1, it destroys the first secret key and moves on to the second host.

There are a number of advantages to employing PRACs. First, if the agent is in control of encapsulating partial results, then this technique can be “applied independently in the design of an appropriate agent application, regardless of the capabilities of the [underlying] agent platform or supporting infrastructure.” [10] Because they are similar to MACs, PRACs are cheaper than digital signatures to compute so some efficiencies in processing may be observed [30]. Finally, the act of destroying the secret keys as the agent progresses guarantees forward integrity. That is, the results up to (but not including) the malicious host cannot be modified without detection.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

An enhancement to this methodology is to equip the agent with an initial key and a one-way function. At the first host, the agent uses the initial key to sign the data collected at that host. Then, it uses this key as an input to the one-way function producing a second key. The first key is destroyed and the agent is forwarded to the next host. To authenticate the integrity of the results, the agent originator uses a copy of the original key and the one-way function to generate the sequence of keys needed to authenticate the data.

2.2.4.2 **Application to Vulnerability Scanning**

As discussed earlier, one of the weaknesses observed with public key encryption is that each host on the network must maintain a private signing key for integrity and non-repudiation purposes. If an attacker compromises a host's execution environment and manages to obtain the host's private signing key, then the attacker can effectively "pose" as that host until the compromise is discovered. It would be beneficial if the host execution environment was relieved of the duty of maintaining a private signing key. The use of PRACs achieves this.

The dynamic creation and systematic destruction of the keys also means that the keys are essentially "one-time use." With PRAC's, the attacker must work at obtaining the signing keys from each agent as it arrives. If the agent is strongly obfuscated, this could take a considerable amount of time. And, if the agent is suitably time-limited, then the attacker may not be able to make use of any stolen keys before the expiration date. The ultimate system would combine PRACs with provably secure encrypted functions.

Due to its lower computation overhead and relative infrastructure independence, the use of PRAC's to guarantee the integrity of partial results may be a significant contribution to a vulnerability scanning system. However, without some other protection mechanisms (for example, code obfuscation), there remains one significant weakness to consider as indicated by [10]. When an agent arrives on a particular host, the host could (if it were malicious) copy all of the keys for the remainder of the agent's itinerary (or the current key and the key generation function). This may or may not be a difficult operation or could be made difficult (that is, with obfuscation). If the agent returns to that host, then the host could use the copied keys to alter and re-sign some of the partial results unless they have been encrypted by some mechanism (the public key of the agent originator, perhaps). Even in this case, the malicious host could still delete some of the results collected and replace them with bogus, signed data. It is assumed here that the agent originator's public key is used to encrypt the data and that all hosts have access to this key. Such activities could be mitigated to some extent through the use of trusted time-stamping services (which introduces additional overhead) or by explicitly forbidding a re-visit to previously called-upon hosts.

2.2.5 Execution Tracing

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

2.2.5.1 **Description of Technique**

Execution tracing [47][48] is a method aimed at *detecting* whether an agent code was executed faithfully by a hosting execution environment. The technique is illustrated in Figure 13.

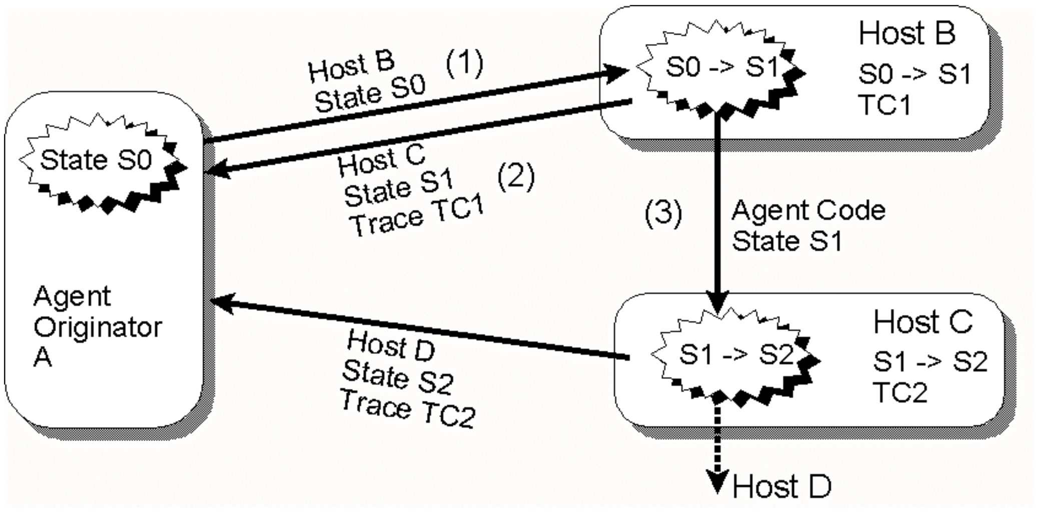


Figure 13
Execution Tracing

An agent originator, A, dispatches an agent (consisting of agent code and state S0) to a remote host execution environment, B (1). It is assumed by [47] that the environment employs a public key infrastructure to allow secure transportation of agents between hosts (that is, no observation or modification of the agent during transit can take place) and that digital signatures are used to achieve integrity and non-repudiation.

The remote host B executes the agent code starting with state S0 (the state that the code ended with at the last host or the initial state if the code was just dispatched). While the code executes, the host produces a trace, Tc, of the execution. This trace is recording of every line of code that was executed by the agent in the order that it was executed. For instructions that modify the program state based on internal program variables, the trace is just an encoding of the operation that was performed. For instructions that modify the program state based on values supplied by the execution environment (for example, an instruction that performs a memory read), the trace also contains the values that have been imported [47].

When agent execution is complete, B signs and stores a copy of the trace for future reference. It then produces a signed hash of the agent final state S1, the execution trace Tc, and the

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

identification of the next host and sends this information to the agent originator[3] who stores it for later reference (2). Finally, the agent propagates its code and new state S1 to host C (3). After all hosts in its itinerary have been visited, the agent returns to the agent originator.
At this point, if the agent originator has reason to suspect that agent execution was tampered with during its travels, it can request a copy of the execution trace from each of the hosts that were visited. Then, starting with the initial state S0 at dispatch time, the originator validates the agent execution on the first host, B, by executing the agent code in accordance with the execution trace provided by B. If a hash of the final state (S1) reached via this process does not match a hash of the final state reported by B, then the execution environment on B did not faithfully execute the agent. Alternatively, if the agent originator produces a hash of the same final state as that found after execution on B, then it begins the process over with the execution trace obtained from host C and initial state S1 from host B until the trouble is found.

Note that the description above is a truncated version of that presented in [47] and ignores a number of steps that were introduced by the author to ensure the integrity and authenticity of the traces and states that are transmitted to the agent originator.

The major drawbacks of this technique, summarized by [10], involve the number and size of the logs that need to be maintained. As well, there is no clear delineation of what constitutes "suspicious" agent results and therefore no clear reason for invoking a trace verification. Finally, the methodology outlined by [47] makes clear assumptions about the availability of a public key infrastructure which may not be valid or practical in all cases.

A recent and interesting approach to address the problem of how to trigger a trace verification is provided by [49]. In this technique, shown below in Figure 14, *every* trace is verified by a dedicated third party known as a verification server (thereby nullifying the issue of when to verify a trace).

[3]In [47], this information is sent to a "receipt" host that may or may not be the same host as the originator.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

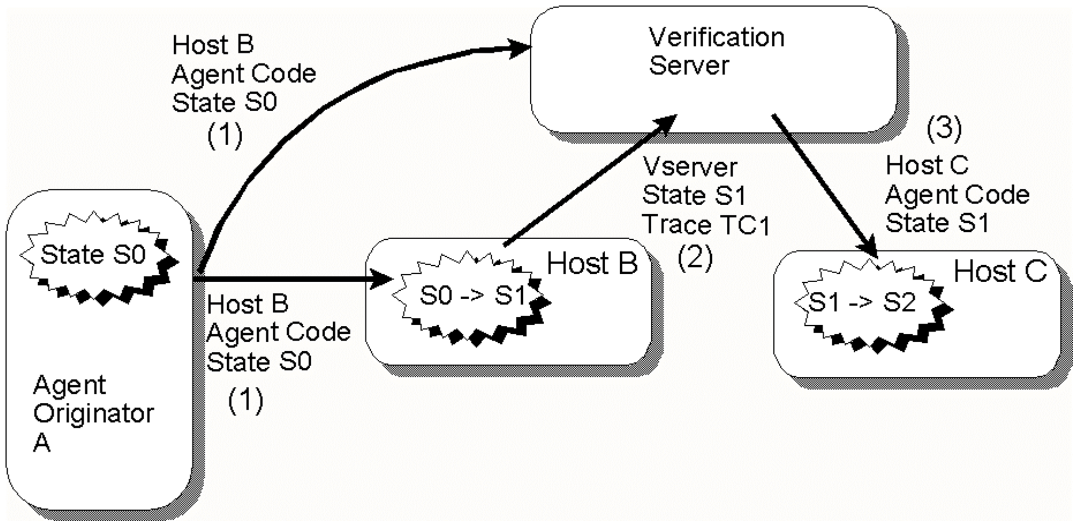


Figure 14
Execution Tracing with Verification Servers

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

In the figure, the agent originator sends the agent and initial state to a remote execution environment on host B. At the same time, it sends a copy of the same information to the verification server (1). When the agent is ready to migrate, host B sends a copy of the final state of the agent after execution on that host to the verification server along with an execution trace (2). The verification server confirms the trace and, if it passes, forwards the agent and state on to the next host (3). The verification server keeps a copy of the agent and state to work with during the next iteration unless there is more than one server in the network. In this case, it may forward the agent code and state to another server. The process is repeated until the agent returns home.

The authors of [49] outline a number of advantages of this new approach. The primary benefit is that each trace is verified immediately so that potential trouble is detected quickly. And, since all traces are verified, the issue of what might be suspicious activity (which would invoke a trace verification) no longer applies. This property also releases each host from having to retain a copy of the execution trace (as in the original methodology) which could be costly if the number of agents on the network is large.

2.2.5.2 **Application to Vulnerability Scanning**

Because it is a method of detecting agent tampering that "runs in the background," there would be no direct effect on vulnerability scanning agents and could be employed in tandem with a scanning system. However, the technique (in its original form) does appear to incur a large overhead due to the storage of trace logs on each host for each agent. While the second approach identified appears to free up these resource constraints, it does so at the cost of freedom and dependence. It breaks the "mobile" agent paradigm somewhat by introducing an intermediate relay point that every agent must visit after it visits each host. As well, it also introduces a single point of failure (which can be mitigated somewhat by replication) which must be carefully observed and secured.

2.3 **Environment Attacks Agent**

2.3.1 **Description**

In this category of security attack, the attacker is an external element that, for the purposes of the present analysis, will accost an agent in transit. There are several general types of attack that can take place [10]. These may include attempts to replay an agent (where such replay attacks can prove beneficial to the attacker), denial of service attacks which simply prevent agent transfer, and attacks on confidentiality through various methods of eavesdropping (which include "sniffing" of network traffic and "man-in-the-middle" attacks

2.3.2 **Preventative Techniques**

The problem of protecting agents against the environment during transit is viewed as having been reasonably well solved through the use of cryptographic techniques [50][2]. Methods to secure

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

the confidentiality of agent code and data whilst in transit include the use of Secure Sockets Layer (SSL) session encryption or local encryption (and signing) of agent code and data before transmittal.

Note that these techniques are often sufficient but not necessarily impregnable. For example, if SSL is not properly enabled, managed, or used, it may be possible to conduct a man-in-the-middle attack. In this scenario, a malicious entity places itself in-between two hosts that wish to communicate and masquerades itself as the receiving host. After it establishes an SSL connection with the initiating host, it initiates an SSL connection with the intended recipient and just relays all data from the initiator to the recipient. However, it is able to observe the agent code and data as it passes through. If implemented properly, cryptographic protections for agents in transit can make it prohibitively expensive for subversive or eavesdropping attacks

2.3.3 **Application to Vulnerability Scanning**

In the perspective of vulnerability analysis agents, it is vitally important to ensure that the data collected by any agent is not subject to observation (or modification) during transit. As discussed, this is adequately addressed by either encrypting the communications session (via SSL) or by encrypting and signing the agent before transmission. We have already seen techniques to allow the partial results to be encrypted incrementally.

Replay and denial of service attacks are not seen as critical attacks in a VA system. In a replay attack, an agent transfer between hosts is “recorded” and then played back at a later time. In such attacks, the benefit to the attacker would be minimal since no useful information or work could be obtained (the traffic stream “recorded” is encrypted). However, if it were possible to easily perform a replay attack, it could be scaled up to perform a denial of service attack by essentially tying down resources on one or more hosts. A denial of service attack generally manifests itself quickly and poses no real threat to the confidentiality or integrity of the data collected by the VA agents.

In the extended Bruce tool [8], when a vulnerability test has been completed by the pollet on a host, the execution environment authenticates and logs into an Entrust PKI directory and retrieves the public encryption key of the deamon or entity responsible for initiating the vulnerability test. It uses this key to encrypt the report generated by the vulnerability scan. The execution environment then signs the report using the host’s private signing key. At submission time, this report is sent back over a clear-text channel.

In [9], the partial results collected on each host are encrypted using the public signing key of the agent coordinator and are signed using the private key of each host. In addition, the agent code and itinerary are signed by the agent coordinator at launch time to protect agent integrity. Finally, when an agent prepares to move from one host to another, it is encrypted using the public key of the intended destination host and signed by the private key of the sending host. The receiving host authenticates the agent upon arrival using the public key of the sending host and

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

then uses its private key to decrypt the agent. In this manner, the agent confidentiality and integrity are maintained during transit.

3 **Conclusions**

In this work, a number of mechanisms were reviewed for enhancing the security of mobile agents in a vulnerability analysis setting. Some of these mechanisms are currently available and applicable and can be used to secure various portions of an agent's transactions with its environment (for example, cryptographic methods, sand-boxing, code signing, PRACs, and some obfuscating techniques depending on the application language). Some techniques require further research or practical application in order to reduce some of the cost of implementing them. These include proof-carrying code, execution tracing, and all forms of code hiding.

No single technique will secure all aspects of an agent's travel and interaction with its environment. It is important to characterize those elements of security that are important to the application and to apply the techniques described above (and other methods not described) in combination to achieve a relative level of security for the agent environment. In a vulnerability scanning application, the most important security elements were the confidentiality and integrity of the reports as well as the integrity of the agent code. It was observed that cryptographic methods could be used successfully to implement these security services (as demonstrated by the analysis of [8] and [9]) but that a certain amount of overhead is assumed for key management. The maintenance of private keys by individual agents or hosts opens up a completely new set of security risks.

As a final note, this work had very little to do with agent based technology although agents are the setting for much of what was presented. Because of the relative novelty of the mobile agent paradigm, much of the effort involved in finding security solutions comes from other fields and domains of research.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

References

[1] Papaioannou T. Mobile information agents for cyberspace - state of the art and visions. In *Proceedings of Cooperating Information Agents* (2000).

[2] Rothermel K., Hohl F., and Radouniklis, N. Mobile agent systems: what is missing? In *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems* (DAIS'97), Cottbus, Germany (September 1997), pp. 111-124.

[3] Hefeeda M, Bhargava B. On mobile code security. CERIAS Tech Report 2001-46.

[4] Karmouch A. Class notes. ELG7187A: Mobile Software Agent Technologies for Telecommunications (Fall 2002).

[5] Mobile Agents White Paper. General Magic, 1996. http://www.diku.dk/topps/activities/plan10/papers/whitepaper.html

[6] Schwartau W., *Cybershock*, Thunder's Mouth Press, New York, N.Y., 2000, p. 189.

[7] *Network and Host Based Vulnerability Assessment*, ISS White Paper, www.isskk.co.jp/customer_care/resource_center/whitepapers/nva.pdf, p.6.

[8] Crocker M. An examination of Sun Microsystem's Bruce application and the mitigation of a confidentiality and integrity weakness suing the Entrust public key infrastructure. Master's Thesis, Royal Military College of Canada (June 2001).

[9] Humphries C., Carver C., and Pooch, U. Secure mobile agents for network vulnerability scanning. In *Proceedings of the 2000 IEEE Workshop on Information Assurance and Security*, USMA, West Point, NY, USA (June, 2000).

[10] Jansen, W. Karygiannis, T. *Mobile Agent Security*. NIST Technical Report, National Institute of Standards and Technology (2000). Available at csrc.nist.gov/mobilesecurity/publications.htm.

[11] Jansen W. Countermeasures for mobile agent security. Computer Communications, Special Issue on Advances in Research and Application of Network Security, (2000).

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

[12] Tan H., Moreau L. Trust relationships in a mobile agent system. In G. Picco (Ed.) Mobile Agents, Lecture Notes in Computer Science, Springer-Verlag, Berlin (2001), vol. 2240, pp. 15-30.

[13] Borselius, N. Mitchell C., and Wilson A. On mobile agent based transactions in moderately hostile environments. In DeDecker B., Piessens F., Smits J., and Van Herreweghen E. (Eds.), Advances in Network and Distributed Systems Security - Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security, Klewer Academic Publishers, Boston (2001), pp. 173-186.

[14] Wang C., Davidson J., Hill J., Knight J. Protection of software-based survivability mechanisms. International Conference of Dependable Systems and Networks, Goteburg, Sweden (July 2001).

[15] Rubin A., Geer D. Mobile code security. IEEE Internet Computing, vol. 2, no. 6 (November 1998), pp. 30-34.

[16] Hashii B., Lal M., Pandey R., Samorodin S. Securing systems against external programs. IEEE Internet Computing, vol. 2, no. 6 (November 1998), pp. 35-45.

[17] Secure Computing with JAVA: Now and the Future. Sun Microsystem's White Paper (2002). http://java.sun.com/marketing/collateral/security.html.

[18] Tschudin, C. Mobile agent security. In Matthias Klusch (Ed.), Intelligent Information Agents: Agent Based Information Discovery and Management in the Internet, Springer-Verlag, Berlin (1999), pp. 431 - 446.

[19] Altmann, J., Gruber F., Klug L., Stockner W., and Weippl E. Using mobile agents in real world: a survey and evaluation of agent platforms. In *Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS and Scalable MAS at the 5th International Conference on Autonomous Agents*, Montreal, Canada (2001), pp. 33-39.

[20] Gong L., Schemers R. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California (March 1998), pp.125-134.

[21] Zhong Q., Edwards N. Security control for COTS components. IEEE Computer Magazine, vol. 31, no. 6 (June 1998), pp. 67-73.

[22] Muffett A. SENSS Bruce - developing a tool to aid intranet security. Usenix ;login: Magazine, (November 1999).

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

[23] Loureiro S., Molva R., Roudier Y. Mobile code security. In *ISYPAR 2000* (4èème Ecole d'Informatique des Systèèmes Parallèèles et Réépartis), *Code Mobile*, Toulouse, France (February 2000).

[24] Kuhn R., Hu V., Polk T., Chang S. Introduction to public key technology and the federal PKI infrastructure. http://csrc.nist.gov/publications/nistpubs/800-34.

[25] Schneier B. *Applied Cryptography - Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1994.

[26] Code Signing Digital Ids for Microsoft Authenticode. www.verisign.com/resources/gd/authenticode/authenticode.html

[27] Internet X.509 Public Key Infrastructure Roadmap. Internet Engineering Task Force. www.ietf.org/internet-drafts/draft-ietf-pkix-roadmap-09.txt

[28] Gutmann P. PKI: It's not dead, just resting. IEEE Computer, vol. 35, no. 8, pp. 41-49.

[29] Simple Public Key Infrastructure. Internet Engineering Task Force. www.ietf.org/html.charters/spki-charter.html

[30] Yee S. A sanctuary for mobile agents. In *Proceedings of the DARPA Workshop on Foundations for Secure Mobile Code*, Monterey CA, USA (1997).

[31] Necula G., Lee P. Proof carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University (Sept 1996).

[32] Necula G., Lee P. Safe, untrusted agents using Proof-Carrying Code. In Vigna G. (Ed.), Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, Berlin (1998), vol. 1419, pp. 61-91.

[33] Appel A, Micheal N., Stump A., Virga R. A trustworthy proof-checker. Princeton University CS TR-648-02 (April 2002).

[34] Neophytos G., Appel A. Machine instruction syntax and semantics in higher order logic. In 17th International Conference on Automated Deduction (CADE-17), Springer-Verlag (June 2000), pp 7-24.

[35] Sander T. and Tschudin C. Protecting mobile agents against malicious hosts. In Vigna G. (Ed.), Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, Berlin (1998), vol. 1419, pp. 44-60.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

[36] Collberg C. and Thomborson C. Watermarking, Tamper-Proofing, and Obfuscation - tools for software protection. IEEE Transactions on Software Engineering, vol. 28, no. 8 (August 2002), pp. 735-746.

[37] Hohl F. A model of attacks of malicious hosts against mobile agents. In *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations* (1998), pp. 105-120.

[38] Kotzanikolau P., Burmester M., Chrissikopoulos V. Secure transactions with mobile agents in hostile environments. In *Proceedings of Information Security and Privacy, 5th Australasian Conference*, ACISP 2000, Brisbane, Australia (July 2000), pp. 289-297.

[39] Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Technical Report #148, Department of Computer Science, University of Auckland, New Zealand (July 1997).

[40] Low D. Protecting java code via code obfuscation. ACM Crossroads, (spring 1998). http://www.acm.org/crossroads/xrds4-3/codeob.html.

[41] Hohl F. Time limited blackbox security: protecting mobile agents from malicious hosts. In Vigna G., editor, Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, Berlin (1998), vol. 1419, pp. 92-113.

[42] CodeShield Java ByteCode Obfuscator. www.codingart.com/codeshield.html

[43] SmokeScreen Java Class File Obfuscator. www.leesw.com

[44] Retroguard Java Obfuscator. www.retrologic.com

[45] Zelix KlassMaster Java Obfuscator. www.zelix.com/klassmaster

[46] CloakWare. www.cloakware.com

[47] Vigna G. Protecting mobile agents through tracing. In *Proceedings of the Third ECOOP Workshop on Operating System support for Mobile Object Systems* (1997).

[48] Vigna G. Cryptographic traces for mobile agents. In Vigna G., editor, Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, Berlin (1998), vol. 1419, pp. 137-153.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

[49] Tan H., Moreau L. Extending execution tracing for mobile code security. Second International Workshop on Security of Mobile Multiagent Systems (2002).

[50] Roth V., Conan V. Encrypting Java archives and its application to mobile agent security. In Agent Mediated Electronic Commerce: A European Perspective (F. Dignum and C. Sierra, eds.), Springer-Verlag, Berlin (2001), pp. 19-33.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image