

Using Security Mechanisms in Cougaar

Richard Feiertag
Cougaar Software, Inc.
rfeiertag@cougaarsoftware.com

Jaisook Rho
Cougaar Software, Inc.
srho@cougaarsoftware.com

Sebastien Rosset
Cougaar Software, Inc.
srosset@cougaarsoftware.com

Abstract

The Cougaar agent architecture provides mechanisms for implementing security of the system and its applications including access control, authentication, authorization, intrusion detection, encryption, and policy management. The paper describes many of the security mechanisms in Cougaar. Each of these mechanisms is separately deployable and configurable. The paper also discusses identification of security requirements for applications, selecting security mechanisms to fulfill the requirements, evaluating the degree to which the requirements are fulfilled, and making trade-offs between security and other requirements such as cost and performance. The resulting process provides the application developer a means for best utilizing the provided security mechanisms.

1. Introduction

As part of the DARPA ALP and UltraLog programs we have been enhancing the Cognitive Agent Architecture (Cougaar) to meet rigorous survivability requirements. We define survivability as having three main aspects: security, robustness, and scalability. Robustness refers to the ability to withstand component failure either accidental or deliberate. Scalability refers to the ability to tolerate major and rapid changes in system load such as might occur in time of war. Security refers to the ability to withstand deliberate information warfare attack such as viruses or Trojan horses. We make the somewhat arbitrary distinction that kinetic attacks, i.e., physical attacks on system hardware, are countered using robustness techniques whereas logical attacks are countered via security techniques. This paper focuses on Cougaar security.

Cougaar has several properties that add to the complexity of achieving security:

Distributed – The system is distributed, consisting of potentially many geographically dispersed interconnected nodes.

Heterogeneous – The system can be implemented with a diversity of platforms including Unix and Windows platforms as well as PDAs and cell phones. The system is Java based and can use most any platform that supports a Java virtual machine, however, some applications may have special requirements that limit the applicable platforms. In addition, the platforms may be interconnected by a diversity of networks ranging from high speed gigabit networks to low speed kilobit networks using a variety of protocols such as direct connections using TCP to store-and-forward message using SMTP.

Agent based – Applications are implemented as interrelated services implemented on mobile agents. Tasks are performed by groups of agents that organize themselves using service discovery.

Loosely coupled – The largely independent services interact mainly using asynchronous protocols.

Large – The system can be very large, incorporating potentially tens of thousands of agents or even larger.

These properties are both a blessing and a curse, providing the basis for redundancy and resiliency through which survivability can be obtained and introducing the complexity that can make survivability difficult to achieve.

2. Cougaar Architecture

A Cougaar society consists of a collection of applications built on top of the Cougaar infrastructure. Cougaar is a distributed, agent-based system built using Java. Each agent provides one or more services defined by the application. Applications are implemented by program units call plugins that communicate with each other by sharing objects in a blackboard. Plugins subscribe to objects by providing predicates that test objects for the presence of selected properties. If another plugin publishes an object to the blackboard that complies with the subscription predicate the subscribing plugins are notified and can access the published object. Agents can also designate that certain objects be shared with other agents, i.e., when one of these objects is published on the

blackboard it is forwarded to the agents of other blackboards.

Multiple agents can reside on a node and can move between nodes. Each Cougar node contains a Java Virtual Machine (JVM) that provides the execution environment for agents as well as services to support the agents and their applications. One of these services is a discovery service that allows plugins to register their capabilities and permits other agents to search for and access those capabilities. Remote services are accessed by publishing special objects, called tasks, in the blackboard of the invoking agent which are then forwarded to the blackboard of the agent providing the remote service. The Cougar infrastructure locates and forwards the object to the blackboard of the remote agent.

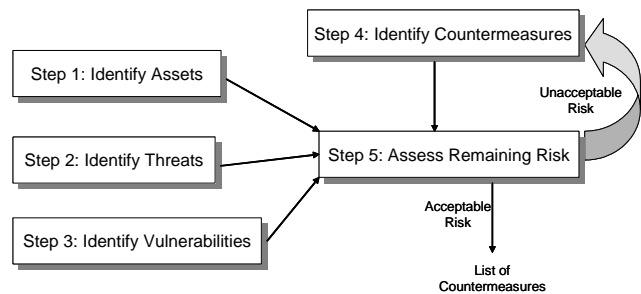
3. Security Approach

The process of assessing the security of a Cougar society is shown in Figure 1. The first step is to define the security requirements of a system, including identifying the assets (data or resources) that need to be protected. Step 2 is to identify the possible threats against the application, i.e., ways in which the assets could be damaged. Third is to identify vulnerabilities of the system, i.e., features of the system that could be used to cause damage to the assets. An insecurity exists in the system only if all three elements (protected asset, threat, and vulnerability) exist. Once the assets that need to be protected as well as the threats and the vulnerabilities have been identified, the rest of the process is iterative. Countermeasures are identified and the residual risk is calculated given the identified countermeasures. More countermeasures can be added until an acceptable level of residual risk is obtained. Countermeasures have an associated cost. The cost of the countermeasures must be weighed against the risk of damage to assets and some balance achieved. This is generally accomplished by selectively adding or removing countermeasures until an acceptable level of risk is obtained at reasonable cost. The result of the above is a set of countermeasures that provide an acceptable level of security.

Without a specific context in which to identify threats and evaluate risks, we cannot carry out the assessment process. Instead we have chosen to identify and implement countermeasures that will be effective against likely threats in operational environments and provide tools and techniques for assessing their effectiveness and computing residual risk. Below are some of the countermeasures being developed.

3.1. Access Control

Cougar provides several mechanisms to control access to resources. The system itself implements a



- Step 1:** *Identify the important assets within the application and the value of each of the assets.*
- Step 2:** *Specify the threat environment of the system by identifying the potential attacks on the system and evaluating the likelihood of those attacks being realized. This will involve understanding who the adversaries are and what their objective may be. The probability of realization may be dependent on the value of the system assets and the cost of exercising the attack.*
- Step 3:** *Identify the vulnerabilities of the application and scenarios for exploiting the vulnerabilities.*
- Step 4:** *Identify countermeasures that limit the ability to exploit vulnerabilities and determine the effectiveness and cost of each of the countermeasures.*
- Step 5:** *Using the data from the above, determine the risk to the system from the identified threats. Determine a cost-effective set of countermeasures that reduce that risk to an acceptable level.*

Figure 1. Assessment Process

variety of resources including Java objects, threads, agents, nodes, tasks, relays, files, etc. Since the system is extensible, applications can implement additional resources. Access control provides the countermeasure that helps protect critical resources from undesirable manipulation. In general, Cougar provides mechanisms to permit or deny a specific actor to operate on a specific resource in a particular manner. Actors can be users (typically acting through browsers), agents, plugins, and external systems. Resources, other than those provided by the host system or JVM, are implemented by designated classes called services. Services define resources and the operations that can access those resources. The implementation of access control has three aspects: enforcement, identity, and authorization and mediation.

3.1.1. Enforcement

Access control must be enforced whenever an actor attempts to access a resource. This enforcement can be done by the class that implements the service or by code interposed between the actor and the service. In order to minimize the amount of security relevant code in Cougar and to relieve the service implementer of responsibility for security, enforcement is usually accomplished by security binders or proxies interposed between the actor and the class or classes that implement the service. Binders and proxies are classes interposed in the architecture between a client attempting to access a resource and the class that implements the resource. These binders verify that the actor has the proper authorization to invoke the requested service. Enforcement decisions are typically simple, being based on the identification of the actor and the identification of the resource or service method being invoked. In such cases, the binders can be generic and table driven requiring only a small effort on the part of the service designer. However, in some cases, enforcement decisions can be dependent on the signature of the method invocation, the system or agent context (e.g., time of day), or the state of the resource (e.g., content of a data base record). These cases may require custom binders or enforcement by the service itself. These cases require careful attention by the service developer. Cougar includes some special purpose enforcement mechanisms such as the Java Security Manager to control access to particular Java Virtual Machine and host system resources and uses Tomcat [1] to enforce access by users to specific URLs via web servers.

An essential aspect of the enforcement mechanisms for all critical services is that they be nonbypassable. That is, access checks always be performed for all invocations of critical services and that the underlying resources that are used to implement the service are properly protected. For example, this means that for a service that implements a database, all the methods that the service provides to access the database must implement proper access checks either in the database service itself or via an enforcement binder. In addition, resources used to implement the database, such as underlying operating system files, must be protected so that they are only accessible to the database service. This might be accomplished using access control to limit access to database files to the database service actor. Similarly, all other resources used by the database service such as caches, buffers, port, and network services must be protected. Although Cougar provides mechanisms to assist in the protection of resources, it is the service designer and developer that must provide the assurance that the service and all its underlying resources are properly protected and nonbypassable.

3.1.2. Identity and Authentication

For the purpose of access control, Cougar supports three types of actors: users, agents, and plugins. Users are identified by their unique user name. When a user attempts to access the system via a servlet, the user is authenticated via a dialog between the web server (Tomcat) and the user's client program (usually a web browser). The authentication is accomplished using a user supplied user name and password or digital certificate and private keys held by the client program or client hardware device. The authentication method used is dictated by system policy.

Agents are identified by their unique agent name and plugins by the class name of the plugin within the agent. Each Cougar thread maintains the name of the agent and the plugin currently executing the thread in a JAAS [2] context. Binders and services can query the JAAS context to securely ascertain the name of the actor invoking the service.

It is often convenient to deal with groups of actors rather than individuals. A group of actors is called a *role*. A role can be a grouping of users, agents, or plugins. Currently in Cougar roles are statically defined but may, in the future, be extended to dynamic groupings such as communities in the future. Some common examples of roles include the group of users that are system administrators or the group of agents belonging to a particular organization.

Resources also have identities. Since many resources are implemented as Java objects, they can often be identified by their Java references. For example, Cougar blackboard objects are identified by their Java references. Other objects, such as files or hosts, can be identified by name. Since most resources are implemented by Cougar services, the service defines the means for identifying the resources. As with actors, resources can be grouped, for example the group of blackboard objects that are tasks.

3.1.3. Authorization and Mediation

Given the ability to intercept all requests for critical access (enforcement) and the ability to reliably identify actors and resources (identification and authentication), the question remains about how to mediate the requested access, i.e., how to decide whether or not to grant the access. Each service must have an authorization policy, i.e., a set of statements specifying whether or not a given request for access should be permitted. In Cougar, these authorization policies are represented as declarative statements expressed in a simple language. For example, the authorization statement

```
A user in role PolicyAdministrator
can access a servlet named
PolicyServlet.
```

indicates that any user in the role PolicyAdministrator is permitted to make requests of the service provided by the servlet named PolicyServlet. Authorization statements

are expressed in the English like form to facilitate understanding by administrators. The precise form and content of each type of authorization statement depends on the service being authorized. Part of designing a service is defining the authorization statements that control access to resources provided by the service although most services will follow common patterns. Cougaar provides editors for composing authorization statements for each service and classes for use by enforcers to mediate access requests. These are discussed more below. Therefore, the service developer must define the resources and the authorization statements to control access to those resources, but for most types of services Cougaar provides the mechanisms to enforce and mediate access control.

3.2. Communications Security

Cougaar provides mechanisms to manage encryption and digital signature of data in transit to protect it from compromise and unauthorized modification. Encryption is used to assure confidentiality of data and digital signature is used to assure integrity of data and to authenticate the data source. The open source Cougaar system does not include encryption or signature software, but standard interfaces [3] are used to allow incorporation of software that implements common algorithms. All messages between agents as well as messages between clients and servlets can be encryption or signed. The use of encryption and digital signatures and the choice of algorithms (if multiple algorithms are available) are controlled via the policy mechanisms discussed below.

3.3. Monitoring and Response

Cougaar provides a framework for monitoring the security state of a society or a portion of the society such as an organization or enclave and taking action to counter a perceived threat or improve security posture. Data is collected from a diversity of sources and analyzed to identify possible attacks or security damage. This analysis can be done manually, semi-automatically, or automatically. Based on the analysis, decisions are made as to preventive or corrective action. The framework consists of the following types of components:

Sensors – These are components, typically plugins, that detect and report a specific type of security relevant event such as invalid digital signatures, unauthorized access attempts, improper service requests, etc. The detection mechanism of a sensor may actually be implemented in a non-Cougaar component such as a network infrastructure component such as firewall or a host operating system. In such cases, the Cougaar component serves as an intermediary, receiving the event data from the external component, possibly formatting it,

and posting it to the blackboard. Sensors post the events they observe in a standard form [4] on their blackboard.

Analyzers – These are components, typically plugins, that subscribe to events from sensors or other analyzers and process the events to reach some conclusion which is posted to the blackboard. The conclusion may be a more abstract event such as the likelihood that some attack has occurred or some component is damaged. Some analyzer may prescribe actions such as moving an agent or changing a policy. Analyzers can fuse or correlate data or may incorporate decision processes such as rules engines or genetic algorithms.

Console Managers – These agents are surrogates for system administrators responsible for maintaining the security of the Cougaar society. Console managers collect data as directed by the administrator and process and prepare the data for presentation to the user as prescribed. The data collected are typically events posted by sensors and analyzers. Console managers can also initiate new or previously defined queries that search for specified events and present the administrator with the results of the search.

M&R Managers – These agents are responsible for controlling and coordinating the activities of the other monitoring and response components. Typically, an M&R manager controls monitoring and response for a portion of a Cougaar society, such as an organization, the hosts on a subnet or enclave, or some community of agents. An M&R manager responsible for a large community can divide its responsibility among several subordinate M&R managers which can, in turn, subdivide their responsibility, if necessary. The M&R managers can deploy, recall, and configure sensors and analyzers and establish communication between those components. In some cases M&R managers subsume the function of other types of components, typically analyzers. For example, an M&R manager may take on the role of decision maker by subscribing to events from sensors and other analyzers and prescribing actions to be taken. Console managers are associated with M&R managers as a convenient point to collect data and initiate searches since the M&R manager has cognizance of all the sensors and analyzers. However, console managers are created and controlled by system administrators rather than by M&R managers.

Actuators – These components perform actions as directed by analyzers and M&R managers. Essentially any component that performs an action can be an actuator and many components have multiple roles. For example, the Policy Domain Managers discussed below serve as actuators for purposes of monitoring and response as well as policy distributors for system administrators. As with some sensors, actuators can be hybrid components having a Cougaar portion that subscribes to action directives and an external portion that performs the prescribed action such as a firewall that filters network traffic.

Currently, Cougar has a limited number of sensors, analyzers, and actuators. It is anticipated that these will be expanded as resources permit and needs are identified. The M&R framework has been designed to be scalable as the number of components increases and more complex demands are implemented.

3.4. Policy Management

Cougar provides a mechanism for the intelligent coordinated control and distribution of policy to components that are policy directed. Policy refers to parameters of components that can effect the operation of that component during operation. Examples of policy have been mentioned above such as authorization statements used in access control, and enabling and configuring of encryption and digital signing for communications security. Policies are initially expressed in a natural language like representation. This is converted by an automated process into a common formal language called OWL [5]. The use of OWL as the representation for all policies allows analysis for consistency and conformance among all policies of the system. At present this analysis is limited but can be improved over time to help avoid problems as administration of the system becomes increasingly complex. The policies are distributed to nodes where they can be queried by components which they control. Policy management has the following components

Policy editor – This is a non-Cougar application that assists a system administrator in constructing policy statements in natural language like form. The policy editor provides a GUI that helps assure that policy statements are syntactically correct and use the proper vocabulary for the services for which they are intended. The policy editor converts the statements into OWL and forwards them to the policy domain manager for further analysis and distribution.

Policy domain manager – This is an agent that receives policy statements from policy editors or other application. The policy domain manager performs analysis of policies to help assure consistency among policies and conformity with society or organizational policy norms. This analysis includes checks for common types of policy errors and compliance with constraints imposed by organizations or current operating conditions. Based on the content of the policy statements, the domain manager then distributes new policy statements to agents and components subject to them. A society will have numerous policy domain managers. Where necessary, the domain managers coordinate their activities and share policy statements. Each domain manager is responsible for distributing policies to a particular set of agents and maintains a database of policies relevant to its set of agents. New policy statements introduced to one domain

manager are automatically shared with other domain managers that have need for them.

Policy node guard – This component operates on every Cougar node. Components that require policy information register with the node guard and provide data about the types of policy statements they require. After registering, components can construct queries about policies. For example, an enforcement component can inquire whether or not a particular actor can access a particular resource via a particular method. By using queries in this way, components can access policy without needing to parse or interpret the OWL representation of policy statements. The policy node guard can perform the interpretation for simple queries. Alternatively, components can choose to receive current relevant policy statements or updates when they arrive, however, this alternative mechanism is not commonly used.

Policy statements are associated with services and are formulated as part of service design. The service designer provides the form and interpretation of policy statements which must be encoded in tables and classes used by the policy editor, domain manager, and node guard. Developing these policy components is a part of developing policy driven services.

4. Countermeasure Application

The mechanisms described above provide a set of tools for building defenses against attack on a Cougar society. The security assessment process described above provides direction for effectively applying countermeasures. The process involves identifying the resources needing protection (security requirements) and ways to determine the quality of the protection (metrics), identifying possible attacks against the protection (threats), selecting and configuring the mechanisms to thwart the attacks (countermeasures), and testing and evaluation the protection afforded by the countermeasures (assurance).

4.1. Requirements and Metrics

It may seem obvious, but the first step in obtaining security is determining what needs to be protected. Oftentimes designers begin to design protection mechanisms without a clear idea about what they are protecting. The purpose of a Cougar society is to fulfill the needs of applications. Although the Cougar infrastructure needs to be protected, infrastructure protection is secondary to protection of the applications. Once the aspects of the applications that need to be protected are identified, one can then determine which aspects of the Cougar infrastructure are needed to support those aspects of the application and need to be protected as well.

In order to illustrate the process of securing an application, we will use an example. The example is an application to provide a simple online ordering system such as the Books-On-Line [6] example in the Cougar open source. The application contains servlets that allow a customer to peruse a collection of items for sale. The customer can then order one or more items by providing a credit card number and a shipping address via HTTP forms. The application attempts to verify the credit card number and shipping address and notifies the distribution center to ship the items if the verification is successful.

To further our example, we will examine a few types of protection that we might want to apply. First, although we want our listing of available products to be generally available, we want to assure that the listing can be modified only by authorized users. The asset being protected is the database of available products and the nature of the protection is that it be modifiable only by authorized users. A derived requirement is that the society be able to authenticate the selected users and that any actions by those users that can result in modification to the product database be protected as well. Similarly, agents that access the product database including the servlets that provide access to the product information must present the data accurately and therefore must execute only the authorized software for accessing and presenting the product database information.

Second, we want to assure the confidentiality of order information. Once an order is placed, information contained in the order should only be accessible to the user who placed the order and users who are authorized to access order information such as workers involved in fulfillment of the order. Furthermore, we would like to limit the access of workers to just that order information they need to do their jobs such as the items ordered and the shipping address. Fulfillment workers should not have access to credit card numbers.

Finally, we would like to provide some guarantees of performance. The Cougar society cannot guarantee delivery of orders or even shipment of orders, since those actions are outside its control. However, the society may be able to guarantee the time required to process an order, order latency, i.e., place a limit on the time from when a user places an order until it is presented to a worker for shipping including the time required to verify the credit card. Alternatively, the society may be able to guarantee some rate of order processing in the aggregate, for example the society can guarantee to process up to 10,000 orders per hour when demand is high.

This last requirement raises the issue of metrics. The first two requirements, the integrity of the product database and the confidentiality of orders, are often thought of in absolute terms. However, such absolutism is probably unrealistic and unnecessary. It is quite reasonable to establish a confidentiality metric, for

example, an unauthorized attempt to obtain order information will succeed less than 0.01% of the time. No security system is perfect and attempts to achieve perfection may lead to solutions that are counterproductive. In our example, attempts to achieve absolute confidentiality might lead us to attempt to deal with threats such as coincident hardware failures that require double or triple redundant hardware to counter and therefore not cost effective for the unlikely damage that could occur.

4.2. Threats and Countermeasures

Consider now the first requirement, integrity of the product database. In the absence of any countermeasures the simplest threat is that an unauthorized user simply invokes a method to modify the product database. We can counter this threat by using the access control mechanism described above. This requires that an enforcer be implemented for the database manager class, that a role be defined for users authorized to modify the database and that appropriate users be assigned this role, and that we implement policy statements to authenticate users in this role and authorize only users with the role to modify the database.

With this basic countermeasure in place we have gone a long way toward assuring the requirement. However, there are still other threats. An attacker might intercept and alter communications between a user authorized to modify the database and the modification servlets to make it change an authorized modification into an unauthorized one. This can be countered by changing the communications policy to require that communications between authorized users and servlets be digitally signed to protect the integrity of the communications. Another similar attack would be to intercept and modify communications between customers and servlets to provide incorrect information to the customer. This can be addressed by changing the communications policy to digitally sign all communications between servlets and customers.

Similar threats are applicable to the second requirement, confidentiality of orders. In this case, an attacker can compromise confidentiality by intercepting communications related to orders between the customer and servlets, between agents processing the orders, and between the agents process the orders and the organization verifying the credit card numbers. Encryption of these communication channels counters these threats. In addition, an unauthorized user could access any databases or files where the order information is stored. Access control must be used to protect stored order data. Fulfillment staff can be authorized to access order items and shipping information, but access to credit card information must be denied. Customers can be

denied access to all order data or may be allowed access to their own order data. Note that in order to permit access to their own order data, customers must be authenticated both when placing the order and later accessing the data.

Countermeasures are not always complete solutions to threats and some countermeasure may introduce new threats. The use of passwords is one means of authenticating a user. However, passwords are often compromised by techniques such as dictionary searches. To counter this threat monitoring and response can be used to detect repeated login attempts with incorrect passwords and either deny or delay further login attempts. Essentially one countermeasure compensates for weaknesses in another countermeasure.

Another use of monitoring and response is to help maintain performance of critical functions. The third requirement identified above calls for a limit on order latency. Such a requirement indicates that order processing is a critical function of the society. A possible threat to such a requirement could be the need to service other types of requests such as complex customer searches or administrative maintenance. Such actions could be threats to requirements even if there is no malicious intent on the part of the customer or administrator. As a countermeasure to such threats, a sensor could be developed to measure order latency and post an event when latency exceeds the required limit. An analyzer could be developed post an event if these long latencies persisted over some period of time indicating an ongoing problem. A decision engine subscribing to such events could decide to take one or more corrective actions such as suspending or deferring complex searches or administrative maintenance in order to make resources available to order processing.

4.3. Assurance

Although there are numerous taxonomies for categorizing security threats, there are no known techniques for methodically identifying all threats that apply to given requirements. Careful analysis and experience are the best means for a thorough identification of threats. Similarly, the ability of countermeasures to effectively counter threats is difficult to assess. Although the ability of specific countermeasures to address specific threats in specific contexts are well studied for some countermeasures, it can be difficult to apply this knowledge to complex systems with multiple requirements and multiple countermeasures whose interaction is not well understood. Different systems operate in different contexts and the quality of a countermeasure in a new context can be difficult to assess. Analysis, by itself, is therefore not sufficient to

obtain assurance that countermeasures function effectively in most situations.

For this reason, comprehensive testing is necessary to help determine if countermeasures are effective in meeting requirements in a variety of operating circumstances and to help determine the residual risk associated with the use of the countermeasures. Countermeasure effectiveness is tested by attacking the system under a variety of operating conditions. Multiple simultaneous attacks are used to test how countermeasures interact. Determining the values of requirement metrics help determine the effectiveness of the countermeasures. Automating the attacks and the measurement of metrics helps expedite the testing process allowing for more extensive testing under varying conditions.

It is possible to test countermeasure effectiveness because countermeasures are intended to defend against specific attacks. Testing is performed by launching the attacks in a variety of circumstances. It is more difficult to perform threat testing. Threat testing involves trying to determine if a system can defend itself against all possible attacks. This is difficult to do because there are no known ways to identify all possible threats. Currently the most common way to perform threat testing is by using red teams. Red teams are groups experienced in attacking systems who use their expertise and experience to identify vulnerabilities in systems and find ways to exploit those vulnerabilities. The red team constructs attacks based on the exploits and launches them against the system to test its defenses. Red teams do not really provide assurance that all threats have been identified, however, they do often identify previously undetected threats and provide some degree of assurance that attackers will not easily be able to identify new threats.

Testing can be performed on a system dedicated to that purpose or on a system in actual operation. The advantage of the former is that any damage done by the attacks is not harmful to system users. Also dedicated test environments allow one to test the system in a wider range of circumstances that may not occur during operation. Such testing may reveal problems that would not otherwise manifest themselves. Testing during actual system operation has the advantage that the testing reflects actual operating conditions, the disadvantage being that the attacks performed may do real harm to the system and its users. However, even if a test attack does harm users it may still be preferable to discover problems in controlled circumstances rather than as the result of a real attack. If an organization does not want to risk harm from a test attack, an alternative might be to operate the society with test instrumentation in place in order to measure the effects if and when a real attack occurs. Most systems on the Internet are attacked routinely albeit using well known threats. The effectiveness of

countermeasures against such attacks should be well understood.

5. Conclusion

Cougaar provides a broad collection of mechanisms that can be applied as countermeasures to attacks to provide system security. However, proper application and configuration of these mechanisms as countermeasures requires a good understanding of what about a Cougaar society requires protection. The first step in obtaining a secure Cougaar society is not configuring mechanisms, it is understanding the security requirements of the society. This is followed by identifying threats and then selecting and configuring the mechanisms that can best serve as countermeasures to those threats. The final step is testing the defenses of the system against attack and measuring the effectiveness of the countermeasures in defending against the threats and in meeting security requirements.

Not all assets of a society are worth protecting and some countermeasures may be more costly than the assets they are trying to protect. It is important to have a means for determining value and the cost of losing an asset in order to make the decision as to how much to invest in countermeasures to protect those assets.

6. Acknowledgements

The work described here has been performed in cooperation with several other organizations. BBN Technologies was responsible for developing the Cougaar core infrastructure software and was directly involved in the design and development of the mechanisms described. The Institute for Human and Machine Cognition (IHMC) of the University of West Florida was the primary designer and developer of the policy management mechanisms. The Intelligent Security Systems Research Laboratory of The University of Memphis designed and developed the client software, GUI, and console manager components of the monitoring and response mechanisms.

This work is sponsored by the Defense Advanced Research Projects Agency¹ under contract MDA972-01-C-0028.

7. References

- [1] "The Tomcat 4 Servlet/JSP Container", The Apache Jakarta Project, <http://jakarta.apache.org/tomcat/tomcat-4.1-doc/index.html>.
- [2] "Java™ Authentication and Authorization Service (JAAS) Reference Guide", Sun Microsystems,

<http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html>.

- [3] Java Cryptography Extension (JCE) for the Java 2 SDK, v 1.4, Sun Microsystems, <http://java.sun.com/products/jce/index-14.html>

[4] "The Intrusion Detection Message Exchange Format". David Curry, Hervé Debar, Feb. 6, 2004, <http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-11.txt>.

[5] "OWL Web Ontology Language Reference", Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, Feb. 10, 2004, <http://www.w3.org/TR/owl-ref/>.

[6] "Books-On-Line, An Advanced Cougaar Tutorial Version 2.0", Cougaar Software, Inc., July 2003, http://cougaar.org/docman/view/php/18/100/BOL_2_Tutorial_V2.pdf.

¹ Approved for Public Release, Distribution Unlimited