SafeBots: A Paradigm for Software Security Controls

Robert Filman and Ted Linden

Software Technology Center
Lockheed Martin Missiles and Space
3251 Hanover Street O/H1-41 B/255
Palo Alto, California 94304
Filman@stc.lockheed.com
415–354–5250

Abstract

We propose a security paradigm in which software security controls are implemented as ubiquitous, communicating, dynamically confederating agents that monitor and control communications among the components of preexisting applications. These agents remember events, communicate with other agents, draw inferences, and plan actions to achieve security goals. Key features of this paradigm are: (1) linguistic mechanisms for specifying agents, security models, and communications, (2) compilation mechanisms that automatically create and install agents as wrappers around existing application components, (3) algorithmic definitions of how agents communicate to increase the security of systems, and (4) a library of agent code fragments, used by the compilation mechanism, to build actual agents. By automating the generation and administration of security agents, we expect to make it cost-effective to install enough redundant agents so that subversion of system software or of some agents can be detected and responded to effectively.

Introduction

We propose a paradigm shift in our approach to software-based computer security controls. In order to guarantee the integrity of software controls, the traditional focus has been on controls that are simple, passive, verifiable, and built into system software. In our new paradigm, software controls are:

- Flexible and context sensitive
- Active in responding to threats
- Reliable through redundant checking
- Incrementally added to existing systems

We believe the first step toward achieving these goals is to wrap conventional software components with programs that analyze communications into and out of applications, appropriately monitoring and controlling these communications. To be effective, such programs need inherent goals, independent processing, communication facilities, and persistence. This combination of features defines *software agents* [Riecken 94, Wooldridge and Jennings 95]. Building on the notions of robots, softbots (software robots) and safety, we call our system SafeBots™. Individual security agent programs are called safebots (note capitalization.)[1]

While there are many challenges facing the SafeBots paradigm, ongoing advances in information technology favor the ultimate success of an agent-based paradigm for security. This paper addresses three key issues that are critical to the ultimate success of a SafeBots approach to security:

1. What can safebots do?
2. How can safebots be protected from subversion.
3. How can ubiquitous, redundant, communicating safebots be created, administered, and controlled so they are cost effective and commercially viable.

Our SafeBots paradigm emerges from lessons learned about previous paradigms and from new technology opportunities.

[1] We emphasize that the work described in this paper is conceptual and is not augmented by actual implementation.

We begin with a background section that summarizes lessons learned and a technology opportunities section that suggests it is time to break away from old assumptions about how to approach security.

Background

In the early 1970s as the first computer networks emerged, computer security became a compelling issue. An early problem was that, while the shared-access operating systems of that era had mechanisms to prevent one user from accidentally interfering with the work of others, none of these mechanisms were effective against deliberate efforts to violate protection boundaries. Several research efforts were launched to remedy this problem. Many applied repair efforts continued through the 70s—and were uniformly unsuccessful. The research community learned that repairing existing commercial operating systems against deliberate intrusions was futile.

Since repairing flaws doesn't work, attention turned to designing multi-user secure operating systems using two approaches:

1. The operating system provides reliable mechanisms that higher level software can use to enforce various security models and policies. Most of this research pursued a generalization of capability-based addressing. [Needham 72, Lampson et al. 76, Linden 76, Neumann et al. 77].
2. An operating system kernel enforces a well-defined security model such as the Bell and LaPadula multi-level security model [Bell and LaPadula 73]

Examples of each of these approaches eventually reached the marketplace in commercial products. IBM's System 38 used capability-based addressing but did not exploit it significantly for security. Security kernels that were certified to enforce security policies also were marketed, but did not achieve much commercial success. The moral was that security features that impact time to market or performance are unlikely to be built into successful commercial products.

By the 1980s, personal computers were becoming commonplace. The personal computer temporarily avoided the problem of shared access by concurrent users, and surrendered all pretense that any effective security control was provided by the operating system. Of course, once personal computers are linked together, the security concerns burgeoned.

Recent work on security controls has centered on encryption, firewalls, and intrusion and anomaly detection techniques. These techniques can be implemented with a manageable level of reliance on software integrity, and the marketplace for all of these security techniques is exploding. (Not all products on the market minimize their reliance on software integrity, and not all are resistant to a determined intruder.)

By the late 80s, it seemed that even the research community had largely given up on the idea of developing software-based security controls that will resist the attacks of a determined intruder. Yet many privacy, security, and availability requirements cannot be satisfied if the only rigorous tools available are hardware isolation, encryption, rigid firewalls, and network-watching intrusion detectors. The key problem with software-based security controls is that it is very difficult to ensure the integrity of software when one is concerned about deliberate efforts to bypass and subvert the controls. A Maginot Line approach does not work. We need ways to build software-based defenses while minimizing the danger that a determined intruder will simply bypass or subvert the controls.

The Technology Opportunities

Several recent trends in computer technology enable new approaches and new paradigms that have not been practical in the past.

- **Distributed systems.** Distribution introduces additional complexity and vulnerabilities that make security more difficult. However, as we learn to manage complexity and use encryption routinely to protect communications between processors, it becomes possible to view distribution as an advantage for security. Hardware isolation and encrypted communications can be used to protect distributed software with differing levels of confidence. Even a complete subversion of controls at one node need not lead to catastrophic failure. With enough redundancy in the controls, it is feasible that no single failure will lead to any significant security lapses.
- **Decreasing hardware costs.** Large amounts processor time and communications bandwidth can now be devoted to security without degrading response. While it has often been assumed that security should not consume more than about 10% of the computational resources, processing is becoming virtually free. In the future, software controls that consume 90-99% of the resources can be very cost effective. (In many current applications, user interface graphics consume 90% of the resources and no one complains about that.) Over time, it will also become increasingly cost-effective to install security controls on dedicated processors where they are more easily protected.
- **High-level protocol standards.** Emerging standards like HTML and CORBA make it increasingly practical for security controls to monitor and understand component interactions. Security controls can be inserted as extensions of well-defined component interfaces, and well-defined interfaces make it feasible

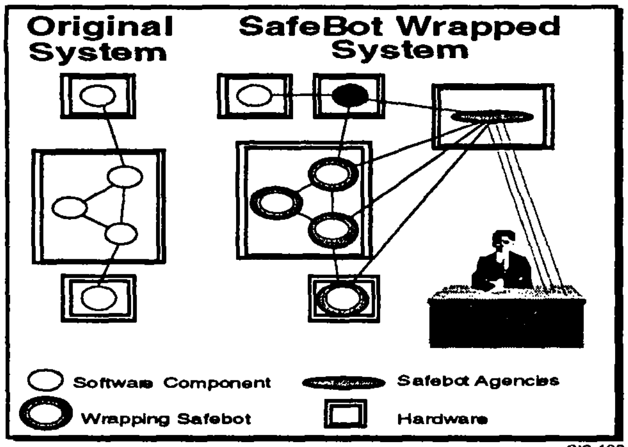

Figure 1: Existing applications evolve into survivable applications by automated addition of safebot wrappers, agencies, and expert assistants

to automate the generation of security wrappers that protect components.

- **Very high level specification languages.** Improving technology for automatically compiling specifications into operational software allows the creation of security systems from high-level requirements. If security controls are going to be pervasive and redundant, then we will have to automate more of the generation, installation, administration, and validation of these controls.

What SafeBots Do

The SafeBots concept is that software security controls become active agents that wrap insecure components, communicate with each other, and are smart enough to adapt their actions to the local and global context. Since safebots are agents, they can be programmed to perform authentication, access control, intrusion detection, or other security controls. In practice, safebots are structured either as wrappers for application components [Genesereth and Ketchpel 94], or as independent safebot agencies [Wiederhold 92] that support the coordination of safebot activities.

Safebots monitor communications by wrapping an application's components. A safebot provides a level of security that is appropriate for the local resource it is protecting, without imposing local constraints on the global system.[2] When wrapping a component, application, or computer, *X*, one replaces it with another component, application, or computer, *Y*, such that *Y* receives all messages to and from *X*, censors or edits them, and passes them on to *X* or to an alternative recipient. Safebots can

- Detect errors or suspicious patterns of activities
- Block inappropriate actions
- Require further authentication before allowing access
- Add to the history of the user, session, or component
- Communicate with other safebots about potential intrusions
- Fix or randomize the duration of the component call to thwart use of timing covert channels, and
- Check that responses do not leak sensitive information.

Figure 1 shows how SafeBots preserves the structure and code of a distributed application while extending it into a highly secure and survivable application. In this figure, some safebot wrappers run on the same nodes as the application components they are protecting, and some run on dedicated processors and intercept all communications to or from a protected component.

Other safebots are independent *agencies* that accumulate information to be selectively shared among safebots. These safebot agencies support communication and collaboration among safebots. Safebot agencies provide common mechanisms for controlled sharing of information about users, computers, sites, system status, normal patterns of behavior, histories of intrusions, recent attack patterns, corrupt software, and the status of other safebots. Some safebot agencies are expert assistants supporting security officers. By being voluntary services with limited trust in other safebots, agencies respond to open networks composed of many independent administrative domains. A given safebot may confederate with different agencies for different purposes [Filman and Linden 96].

Safebot agencies support different security functions; for example, different agencies will provide services that support:

- Authentication of both users and services
- Security status monitoring
- Behavior profiling.
- Rapid communication about known attack patterns, for example, safebot agencies may disseminate dynamically updated information about known viruses.
- Reasoning about the trust to place in communications from other safebots.
- Security officers.
- Security administrators.

Figure 2 illustrates some of the potential safebot agencies. As an example of the additional flexibility that can be achieved with safebot agencies, consider their potential role in supporting very flexible user authentication. Conventional authentication is usually a rigid, static decision

[2] With networks that cross many independent administrative domains, security mechanisms based on imposing policies on others are doomed to failure. With a distributed approach to security, one can put walls and gates in one's own territory and cooperate with (but not completely trust) well-behaved neighbors.

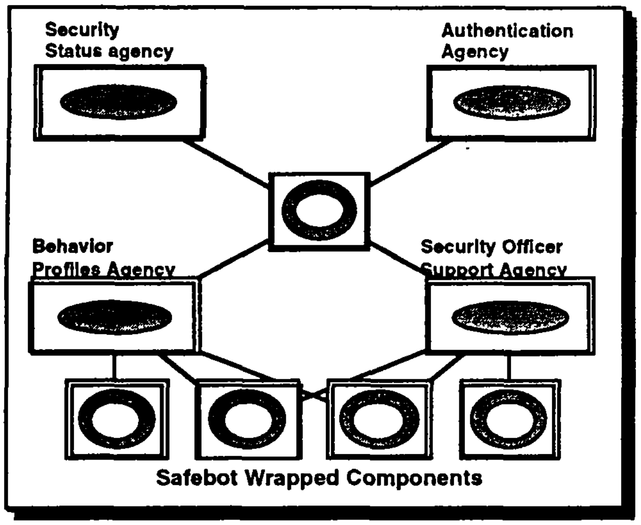

Figure 2. Safebot agencies support communication and collaboration among safebots.

removed from supporting context like the user's location, recent terminal idle time, and the session's recent history of anomalous or suspicious actions. An authentication agency, in addition to storing the information needed to authenticate a user at a remote site, can collect reports about user actions and dynamically determine a confidence level in the user's identity. This information can be shared as a user connects to multiple sites. A safebot protecting a critical resource may check with the user's authentication service before granting a request for especially sensitive information, and then may demand redundant authentication. In one approach to reauthentication, the authentication service maintains a list of user-provided memories that no one else is likely to know. The advantage of this approach is that it requires no special hardware and can be used when the user is at a location where authentication hardware is missing or broken.

Protecting Safebots from Bypass and Subversion

SafeBots builds on, complements, and extends the security provided by encryption mechanisms. We assume that encryption protects safebot-to-safebot communications from eavesdropping and spoofing. The communications of the application programs being protected may or may not already be encrypted. If they are, safebot wrappers monitor the communications before encryption and after decryption. If the application communications are not encrypted, safebot wrappers are a convenient way to add encryption, and safebot agencies are a way to support key management. Encryption can also help sequester application components and prevent the safebot from being bypassed.

The overall security controls enforced by safebots must continue to function reliably even when some of the safebots or the operating system underneath them been subverted. In a distributed system, approaches for dealing with this threat include running safebots on dedicated hardware, employing continuous mutual vetting of distributed safebots, reasoning about the level of trust to place in communications from other safebots, and isolating rogue safebots.

The SafeBots System

Our vision of the SafeBots system (as a software system) is fancifully illustrated in Figure 3. The system has three major parts: **OntoSec**, a language for describing safebot specifications, component behaviors, and inter-safebot communications; **Swathe**, a compiler of specifications and library components into safebot wrappers, stand-alone safebots, and installation scripts, and **SecLib**, a library of reusable safebot components and code fragments. SecLib components come with both code and an OntoSec description of what the code does and how it is to be knit into a safebot. (Some SecLib elements are purely OntoSec description, useful when Swathe knows how to expand such a description by itself or when the description expands out into other defined components.) SecLib has two parts: (1) generic components useful in any application involving wrapping agents and (2) components that are particularly suited for security algorithms. Examples of generic components include inter-safebot communication, database mechanisms for long-term memory and short-term transactions, pattern-matching, and generic wrappers for particular protocols (e.g., WWW and CORBA). Examples of security components include particular implementations of authentication mechanisms, predefined agencies, algorithms for computing trust factors and intrusion detectors.

The user of SafeBots does two things:

- **Adds components to SecLib.** These are code and OntoSec descriptions of that code.
- **Develops inputs for Swathe.**

Swathe, given (1) an OntoSec specification of the desired goals and behavior of the new safebots, (2) an OntoSec specification of the interface of the to-be-wrapped applications, (3) a SecLib of appropriate code fragments, and (4) target information about the locations of the to-be-wrapped applications, produces (1) a set of safebots (both wrapper agents and agencies) and (2) a script for performing the wrapping and installing the safebots. Executing the script performs the wrapping.

The application is now more secure. The systems generated by Swathe scripts can span several applications on multiple machines and may be created piecemeal over time.

We discuss each component in more detail below.

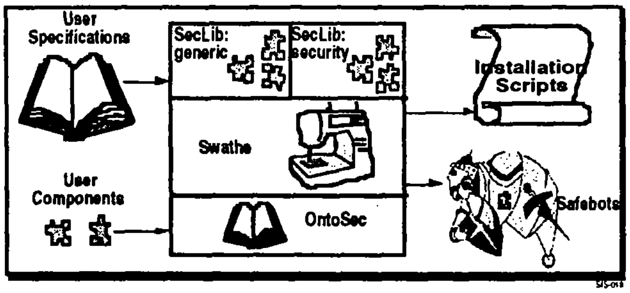

Figure 3: Swathe takes user specifications written in OntoSec, combines them with the library of security components and produces safebots and installation scripts for these safebots.

OntoSec

Our premise is that safebots communicate. Our subpremise is that we will generate safebots from a specification of their desired properties. We use a common language for both safebot specification and safebot communication. We call this language OntoSec, for "ontology for security." OntoSec is a language for representing security requirements, specifications, goals, actions, events, and knowledge of agents. For example, OntoSec can describe protocols; the security properties of resources and components; the privileges of users and sessions; events, actions to be taken on events, and semantic bindings for implementing those actions; and histories of users and systems. OntoSec provides a vocabulary for specifying the security properties to be enforced by safebots and for safebots to communicate with each other and with security personnel.

Method	Item	Category
Person	Class	e.g., Jane_Doe
Event	Class	Something that happens
History	Class	A record of a session or a person
Action	Class	A program to be run in response to events
Session	Class	A login, associated with net operations
Resource	Class	A particular file, database or machine
Operator	Class	An application program
May	Relation	Permission
Item	Relation	A data item within a resource
OnMachine	Function	An operation on a particular resource
HasPrivilege	Relation	Connects privileges with persons or sessions
Knows	Modality	Expresses a safebot's or person's knowledge
Goal	Modality	Expresses the goal or policy of a safebot
Probability	Modality	Expresses probability of an assertion

Figure 4. By providing primitives for the conceptual entities of the security domain, OntoSec enables both formal specification of systems and inter-safebot communication.

Important dimensions of OntoSec are that:

- It is expressive enough for safebots to express their policies, status, knowledge, beliefs, and concerns. It must support safebots in determining the level of trust to place in the messages and requests of other safebots.
- It is directly computable; that is, we want a system that infers the consequences of a collection of security statements in a reasonable amount of time.
- It provides a way of unifying programmatic behavior with reasoning.

Figure 4 lists some typical classes, predicates, and imperatives that need expression in OntoSec. Security systems must deal with people, things that happen, both currently and in the past, actions to be taken on particular events, and the properties of individual applications. The ontology must be able to discuss permissions and obligations, the physical configuration of elements, and the knowledge of individuals and their goals and deal with probabilistic and evidential reasoning. Figure 5 shows typical OntoSec statements in an informal logic, including examples of "user predicates" (e.g., TrustedFriends), permissions for a particular component, the consequences of allowing execution of an unbounded program, and actions to be taken on events (e.g., warnings on repeated password failures and blockages after raised suspicions). Note that the language has "second order" or "modal" aspects, in that it expresses notions such as goals, knowledge, and probability.

This notion of ontological specification of desired behavior is an important theme in current AI research [Neches et. al. 91]. Examples of ontological approaches to security include Yialelis *et. al* [Yialelis et. al. 96] and the deontic logic work of Bieber and Cuppens [Bieber and Cuppens 93]. An important element in the generation of communicating intelligent agents is an appropriate underlying communication protocol; KQML [Finin et. al. 94] is one such language.

Wrapping with Swathe

SafeBots is based on the wrappability of applications and components. We assume that components to be protected (1) are specified—that is, have a well-defined, formally representable interface, (2) can be sequestered—that is, placed where intruders cannot invoke them directly, and

```
John ∈ TrustedFriends (LockheedMartin)
May(p,Write,DB42,x,y) → May(p,Read,DB42,x,y)
May(p,Read,item(DB42,Salary(q)) →
   (p=q | WorksFor (p,q))
May(p,s,OnMachine(k, Shell), x, y) &
UserProgram(m) →May(p,x,OnMachine(k, m), x, y)
Owner(p,r) → ∀m.May(p,m,r,x,y)
FailedPasswordTries(s,h,r) > 5 →
   Notify (SessionHolders (s), PasswordHacking(s)) &
   ∀r x y. ~May(User(s),s, r, x, y)
Goal(Suspicious(p) → ~May(p,r,a,x,y))
```

Figure 5. Because OntoSec provides a language for expressing security concepts, SafeBots systems have a richer environment of behaviors and responses

(3) can be substituted—that is, a replacement component can be introduced into the system in their place. This replacement component supports the specified interface, performs whatever security actions are associated with the call, and invokes the sequestered, original function to do the actual work. Examples of wrapped components range from network proxy servers through UNIX executable shells on to tracing in Lisp.

Manual wrapping is labor-intensive, cumbersome, error-prone, and inconsistent. We argue for the need for tools that perform such wrapping automatically. Such a tool (which we call *Swathe*) takes as inputs:

- The interface definitions of the application components
- OntoSec specifications of desired security properties
- A library of security algorithms and safebot code fragments (SecLib)
- The physical organization of the system (e.g., locations of existing applications)

and produces

- A wrapped application or component that conforms to the specified security properties and
- An installation script for that wrapped application.

Note that Swathe is not dealing with the semantics of application component interfaces—security programmers will write SecLib routines that can do things with the information content being passed. Rather, the automatic programming of Swathe adjusts the safebots code to deal with the syntax of communications—a more tractable and quite useful activity.

An important element of this scheme is the existence of SecLib—Swathe works primarily by selecting appropriate elements from this library and coherently knitting them together. The SafeBots algorithms discussed above would be realized from such components.

When a safebot intercepts a method invocation to or from the wrapped component, Swathe makes additional parameters available to the safebot. These parameters identify the calling session, its security context, and the responsible human source of the call. (Additional parameters like these are already passed by CORBA remote procedure call implementations such as Orbix.)

SecLib

SecLib is an extensible collection of algorithms, mechanisms, and safebot code fragments that understand OntoSec and can be automatically assembled into safebots. These fragments enable safebots to:

- Sense and evaluate their environment to detect security threats
- Understand and reason about OntoSec specifications
- Communicate with other safebots
- Reason about actions to best enforce security policy in the current context
- Reason about communications received from other safebots. (For example: Have they been subverted? What information should I send them? Should we collectively ostracize them? How does their communication affect my understanding of my context?).

Additional safebot fragments implement specific security algorithms. They focus on redundant user authentication, data aggregation, statistical analysis, access control, denial of service due to system overloading, and other specific threats and controls.

Swathe weaves these safebot fragments into safebots capable of enforcing OntoSec specifications for the application component around which they are wrapped.

Safebots dynamically form federations, joined by interest in the behavior of particular users, systems, or sessions. They check on each other and evaluate the trust they place in communications from other safebots. Since safebots are created by the owners (or "partial owners") of components, SafeBots technology supports the realization of systems embodying multiple, overlapping administrative concerns.

Genetic diversity

By including multiple versions of algorithms and fragments in SecLib, we enable Swathe to introduce "genetic diversity" into its space of wrapped organisms. Thus, a hole in one particular implementation of a component does not render vulnerable every user of the semantics of that component. A diverse ecology of security mechanisms is less vulnerable to a single-disease catastrophic failure than is a monoculture of identical organisms. Similarly, a system composed of security elements that trust each other less than completely and whose "genetic code" varies is less exposed to a single point of failure, much as biological systems use multiple immune responses to protect against a variety of parasites However, aside from work on genetic algorithms, it is difficult to get interesting, complex computer programs to evolve on their own.

Research Concerns and Potential Limitations

While safebots have many advantages for security, they also have disadvantages—especially in the near term:

- Safebots wrap only application components that have well-defined application program interfaces (APIs), specified in a supported interface definition language (IDL). Applications with complex GUI's or interpreters (e.g., shells, programming environments) are not good candidates for wrapping.
- We need to determine if safebots can detect subversion of the lower level software running beneath another safebot that they have trusted in the past.
- Initially, safebots will make security administration more complex. It will be difficult for the average security officer to understand everything that is going on. Configuring safebots to check on each other will be complicated. Eventually, the benefits of redundancy, high-level specifications, and visualization will make the security officer's job easier, but it will be some time before we achieve enough redundancy to cover mistakes in administering security.
- Safebots themselves can become a source of catastrophic failure. Subverting a safebot could become a way to attack systems, and inept security designers could design safebots that reduce rather than enhance survivability. SafeBots is designed so safebots can check on each other and limit their trust in other safebots. We need to determine the extent to which these mechanisms are practical.
- Safebots will use computational resources and can degrade response time. At a time of crisis, heightened safebot activity could tie up a system just when it is most needed. Assigning safebots to run on their own hardware is a partial solution to this concern. Eventually, safebots can also reason about the effect they are having on performance. Faster hardware and careful design are also key to long-range mitigation of this concern.

Conclusion

SafeBots is a vehicle for experiments with the cost-effectiveness of redundant security agents distributed pervasively throughout applications. The long term goal of SafeBots is to make defensive controls dramatically less expensive and force intruders to breach redundant barriers—turning the advantage to security defense and fundamentally changing the balance between penetrators and security personnel.

References

[Bell and LaPadula 73] D. Bell and L. Lapadula, Secure Computer Systems, Air Force Electronic Systems Division, ESD-TR-73-278, Nov. 1973.

[Bieber and Cuppens 93] P. Bieber and F. Cuppens, "Expression of Confidentiality Policies with Deontic Logic," in J. Meyer and R. Wieringa (Eds.) *Deontic Logic in Computer Science: Normative System Specification*, Wiley, 1993.

[Filman and Linden 96] R. Filman and T. Linden, "Communicating Security Agents," *Proceedings of the Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Stanford, CA, June 1996, pp. 86–91.

[Finin et. al. 94] T. Finin, R. Fritzson, D. McKay and R. McEntire, "KQML as an Agent Communication Language," *Proc. 3rd Int'l Conference on Information and Knowledge Management*, 1994.

[Genesereth and Ketchpel 94] M. Genesereth and S. P. Ketchpel, "Software Agents," *Comm. ACM 37*, 7, 1994, pp. 48–53.

[Lampson et al. 76] B. Lampson & H. Sturgis, "Reflections on an Operating System Design." *Comm. ACM, 19*,5. May 1976, pp. 251-266

[Linden 76] T. A. Linden, "Operating System Structures to Support Security and Reliable Software." *Computing Surveys*, 8,4. Dec. 1976, pp. 409-445.

[Neches et. al. 91] R. Neches, R. Fikes, T. Finin, T. Gruber. R. Patil, T. Senator, & W. R. Swartout. "Enabling Technology for Knowledge Sharing," *AI Magazine, 12*, 3, 1991, pp. 16–36.

[Needham 72] R.M. Needham, Protection syhstems and protection implementations. AFIPS Conf. Proc., 1972 FJCC, AFIPS Press, Nontvale, NJ. 41, pp 571-578.

[Riecken 94] D. Riecken, Guest Editor, "Special Issue: Intelligent Agents," *Comm. ACM 37*, 7, 1994.

[Wiederhold 92] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *IEEE Computer 25*, 3, 1992, pp. 38–49.

[Wooldridge and Jennings 95] M. J. Wooldridge and N. R. Jennings, "Agent Theories, Architectures, and Languages: A Survey," in M. J. Wooldridge and N. R. Jennings (Eds.) *Proc. ECAI-94, Workshop on Agent Theories, Architectures and Languages*, Springer-Verlag Lecture Notes in Artificial Intelligence-890, 1995, pp. 1–39.

[Yialelis et. al. 96] N. Yialelis, E. Lupu, and M. Sloman, "Role-Based Security for Distributed Object Systems", *Proc. IEEE Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1996.