STATE-BASED SECURITY POLICY ENFORCEMENT IN COMPONENT-BASED E-COMMERCE APPLICATIONS*

Peter Herrmann, Lars Wiebusch, and Heiko Krumm

Universität Dortmund, FB Informatik, LS IV, 44221 Dortmund, Germany

Peter.Herrmann@cs.uni-dortmund.de, lars-wiebusch@web.de, krumm@cs.uni-dortmund.de

Abstract Software component technology supports the cost-effective development of e-commerce applications but also introduces special security problems. In particular, a malicious component is a threat to any application incorporating it. Therefore wrappers are of interest which control the behavior of components at run-time and enforce the application's security policies. The wrapper of a component monitors the component behavior at its interfaces and checks its compliance with the security behavior constraints of the component's employment contract. We propose state-based security policy definitions, report on their suitable design, and clarify their employment by means of a component-structured e-procurement application.

Keywords: Security policy enforcement, component security, security wrappers.

1. Introduction

The approach of component-structured software envisages applications composed from cost-effective components. The components are supplied by different developers and are offered to a growing community of customers on an open market (cf. Szyperski, 1997). By selection, configuration, and customization of components powerful applications can be built which are tailored to the special needs of single customers. The benefits of the component approach, however, are accompanied by a series of technical problems. The architecture of component-structured application systems really extends the conception of distributed object-based applications. In particular, it imposes new security aspects since

*To appear in the Proceedings of the *2nd IFIP Conference on E-Commerce, E-Business, and E-Government (I3E)*, Lisbon, October 2002, Kluwer Academic Publisher it introduces new principals and roles. In addition to users and application system owners, also component vendors and host providers have to be considered. On the one hand they introduce their own security objectives. On the other hand, they introduce new types of threats since in general the different principals cannot trust each other to full extent. Considering that enterprises are increasingly dependent on their information systems, the security of the applications is of growing importance. Therefore approaches are of interest which support the security of those component-structured application systems incorporating code obtained from not necessarily fully trusted sources.

Of course, the composition of applications from various components causes not only security problems. Among other properties, in particular it is essential for the functionality of an application that each component acts in accordance with its specifications. Therefore the approach of software components refers to the employment of explicit contracts. Each component integration shall be accompanied by a contract which is legally binding and describes agreed properties of a component and, in particular, its interface. According to Beugnard et al., 1999, a contract consists of four parts specifying the structure of a component interface (i.e., methods, input and output parameters, exceptions), the desired behavior of the component and its environment, synchronization aspects, and quantitative quality-of-service properties.

Our overall approach is also based on contracts. A component contract has to contain a description of the security-relevant behavior with which the component's execution is assumed to correspond:

- At design time, the structure of the system is analyzed in combination with the behavior descriptions of its components in order to prove that required security properties of the system hold if each component will act in accordance with its contract.

- At run-time, the consideration can focus on the components. For each component it is of interest that its actual behavior in fact is conformable with its contract.

We assume that malicious components or compromised code (e.g., by virus or Trojan horse infection) will result in behaviors which diverge from the descriptions of the contract. Therefore the component behavior is controlled at run-time by means of wrappers. A wrapper monitors the interfaces of a component. It detects the interface events and checks their compliance with the security behavior constraints of the component's employment contract. In case of policy violations components are blocked and the application administrators are notified.

The approach is introduced in Herrmann and Krumm, 2001, which reports on the architecture and management of wrappers. Moreover, it describes how enforcement functions for state-based policy definitions are provided in wrappers and outlines a corresponding Java Bean implementation. Since detailed control functions can cause substantial overhead, furthermore trust adaptation is introduced. The control functions of a wrapper are dynamically adapted to that level of trust, the component currently has in the eyes of the application owner. Additionally, Herrmann, 2001, concentrates on the trust management aspects of our approach describing a suitable information system infrastructure and its support for component procurement decisions.

In the sequel we will report on application-oriented aspects of run-time component security policy enforcement. From an application owner's point of view the security policy of a component shall help to discriminate between desired and malicious component behavior in order to expose compromised components threatening the application and the assets managed by it. In principle, this objective is best supported if the policies specify the desired behavior of the component in detail. Then, however, the policy implementation in the wrappers would have the character of reference implementations and the efforts for the development of the policies and their implementations would be comparable to component development. This aspect as well as performance issues plead for the employment of more abstract policies. Therefore policies are of interest which can be defined under abstraction from the detailed component behavior and which concentrate on the essential application contributions of the component. Additionally, we consider that the design of suitable policies is strongly related to the vulnerabilities of the application and that the vulnerabilities depend on the application's component architecture as well as on its application functions. Dealing with these requirements we analyze an example application, recognize the general suitability of state-based policies, and identify helpful policy conceptions and patterns.

In more detail, we specify state-based policies by state transition systems and apply the temporal logic specification language cTLA (Herrmann and Krumm, 2000a) for the modular definition of behavior constraints. The example application is a typical e-business application. We study a component-structured application which supports a traditional shop and which emerges to an e-business application of type business-to-business by integration of e-procurement components.

2. Related Work

The security problems of the integration of non-trusted components are related with security of migrating code. With respect to that various approaches were recently developed in order to protect host computers against attacks by mobile programs. The methods mainly focus on control flow safety, memory safety, and stack safety (cf. Kozen, 1999). Besides of isolating security-critical operations in a protected system kernel (e.g. Bershad et al., 1995) and using cryptography for the transit of code, code instrumentation gained attraction in the last years. Here, machine code is altered in a way that critical operations can be analyzed before or monitored during the execution of the code in order to detect attacks. An example is software fault isolation (e.g., Wabbe et al., 1993) where non-trusted code is executed and monitored in a safe system part where it cannot cause damage.

Another code instrumentation-based approach is pursued by Schneider, 1997, who models policies formally by so-called security automata. Moreover, a security automaton can be used to enforce a policy by simulating it simultaneously to the execution of the code. The code is only permitted to perform an execution step if that corresponds to a transition of the automaton. The automata based enforcement extends the early approach of state dependent security specifications (cf. Biskup and Eckert, 1994).

Language based security is another kind of code instrumentation. Here, special security-related information about migrating code is obtained during parsing or other program analysis'. The user utilizes this information in order to check the code for compliance with his security policies. An example is the Java byte code verifier which proves Java byte code for type correctness and other security-related properties. Another method is proof carrying code (cf. Kozen, 1999) which enables formal program verification. The program developer annotates the code with a formal specification (e.g., pre- and postconditions of functions or loop invariants) and hands this information over to the user who proves the code formally. Examples for utilizing proof carrying code are the touchstone compiler (Necula, 1998) and the efficient code certification (Kozen, 1998). Moreover, this method was used for more specialized verification purposes as type checking (Morrisett et al., 1998; Tarditi et al., 1996) and information flow analysis (Ferrari et al., 1997; Myers and Liskov, 1998).

Since the information used for code verification is produced by the code developer, it may be distorted in order to mask malicious code. Thus, one has to check that the program complies to the additional in-

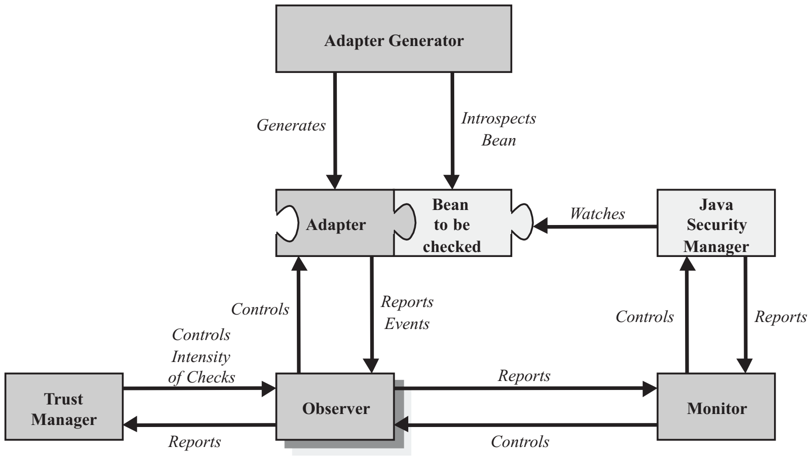

Figure 1. Security Wrapper Architecture

formation used for verification. Here, the concept of generic software wrappers proves helpful. In this approach a program is extended by a software checking the code execution during runtime for security properties. Generic software wrappers are used with firewalls and intrusion detection (Avolio and Ranum, 1994; Goldberg et al., 1996; Monroe, 1993). Moreover they can also be applied for protecting component-structured software from malicious system calls (Fraser et al., 1999).

As our approach, Khan et al., 2001, extend component contracts by security aspects. Unlike us, however, they concentrate on the modeling of requirement-assurance relationships between components. The model has a relatively simple structure and does not represent behavioral properties. Thus, it cannot deal with detailed enforcement policies.

3. Security Wrapper Architecture

Security wrappers (Herrmann and Krumm, 2001) are a useful means to enforce security policies in component-structured software. The interface behavior of non-trusted components is observed and checked for compliance with the security objectives described in the component contract. Figure 1 depicts a wrapper implementation (Mallek, 2000) for component-structured systems based on Java Beans. The system consists of adapters, observers, an adapter generator, a monitor, the built-in Java Security Manager, and a trust manager. Each scrutinized bean is wrapped by an adapter component discerning all events passing the bean interface. Moreover, the adapter may seal the bean by blocking all events if the bean is regarded malicious.

The compliance checks are performed by observer components. An observer simulates a formal specification modeling a security objective defined in the contract of the bean in question. If an adapter discerns an interface event, it blocks the event temporarily and notifies the observers which check if the event is in compliance with the specifications. If all observers accept the event, it is released and forwarded by the adapter. Otherwise, the application administrator is notified and the adapter blocks the bean.

Adapters are created automatically by the adapter generator which utilizes the Java inspection mechanism to detect the event structure of a bean. The monitor acts as the interface to the application administrator. The Java Security Manager is used to prevent hidden data channels of a wrapped bean by permitting only events which pass the corresponding adapter.

The trust manager may be used to reduce the monitoring expenditure depending on the experience, other users gained from the bean. It is linked to a trust information service (Herrmann, 2001) which stores trust values (cf. Jøsang and Knapskog, 1998) of registered beans according to the amount of positive resp. negative evaluation reports. In intervals the trust manager retrieves the current trust values of the observed beans and decides about full observation, spot checks, or complete removal of the adapters. Moreover, the trust information service informs all interested trust managers about reported malicious behavior. Thus, often a trust manager may cause an adapter to seal a malicious bean before it does any harm.

4. Security Policy Patterns

A malicious component may easily spoil a component-based application by performing confidentiality, integrity, availability, or non-repudiation attacks. With respect to confidentiality the flow of data may be changed in a way that a data unit is forwarded to a component granting access to humans who are not allowed to read the data. This threat is particularly relevant in distributed component-structured systems since the components reside on various network-connected computers with different user-access policies. Besides of data flow attacks, confidentiality may be attacked by utilizing hidden channels. The illegally forwarded information is either concealed in transferred data (steganography) or in the order, number, or execution time of interface events between components. Integrity attacks modify application functionality and information by incorrect component operations, by malicious data base updates, and by manipulations of application configuration parameters. Attacks on the availability of components may be performed in two ways: At first, a service provided by another component is called very often. In consequence, the called component cannot serve other components anymore (denial-of-service attack). At second, a component may block a partner temporarily or continuously by refusing to perform a desired interface action. While waiting for this action, the partner cannot serve other components. Finally, with respect to non-repudiation a component vendor may later deny that the component has triggered or received certain interface events.

To protect an application from component attacks, one has to define security objectives and to enforce corresponding policies. The policies constrain the component behavior in a way that attacks are either prevented or made more difficult. To avoid a confidentiality attack, the flow of data between two components must be restricted in order to forward data only to components which prevent reading by not authorized persons. The danger of hidden channels may be reduced by preventing non-deterministic interface behavior (cf. Zöllner et al., 1998). Therefore a corresponding security policy defines a functional dependency between forwarded data units and previous events to avoid steganography. Other policies restrict the order, number, and execution time of events. Security policies counteracting integrity attacks restrict the execution of interface events. They constrain arguments of events and apply plausibility checks. In order to prevent denial-of-service attacks, a protecting security policy may schedule a minimum waiting time between two service requests. Thus, the called component gains some time to serve other components. With respect to blocking other components, a security policy may require that a desired event is executed within a maximum waiting time. The guarantee of non-repudiation is difficult since legally binding proofs of event executions are necessary. A step towards such a proof is the incorporation of an independent trusted third party providing for a logging service and a security policy enforcing the logging of interface events. Log-entries contain digital signatures identifying the originating component.

The security wrappers introduced in Sec. 3 can only check the events at the interface of a component but not the internal attribute settings and internal events. Therefore, enforceable security policies concentrate on the interface behavior. Thus, we can define four basic policy patterns which correspond to the four basic aspects of component interfaces:

- **Enabling condition**: The enabling of interface events and the argument values of the events are constrained.

- **Enabling history**: The enabling conditions of interface events depend on the context of preceding interface events.
- **Minimum waiting time**: Interface events may only be executed if some minimum waiting time periods elapsed since preceding events.
- **Maximum waiting time**: Interface events have to occur before a maximum waiting time expired since preceding events.

These four patterns serve as a basis for more specific policy patterns which are directly devoted to the security objectives listed above. With respect to confidentiality we use the following patterns:

- **Data flow access**: A data unit may only be forwarded to a component if a corresponding read access permission exists.
- **Data flow history**: A data unit may only be forwarded in the context of certain preceding interface events.
- **Hidden channel functional dependency**: A forwarded data unit depends on previously transferred data according to a data dependency function.
- **Hidden channel enabling history**: The enabling condition of an interface event and its arguments depend on the context of preceding events according to a occurrence dependency function.
- **Hidden channel execution time**: An interface event has to be executed after a preceding interface event within a certain time period.

Patterns enforcing integrity security objectives are listed in the sequel:

- **Integrity enabling condition**: The enabling conditions of interface events and their arguments are constrained in order to guarantee plausible component interaction.
- **Integrity enabling history**: The enabling conditions of interface events and their arguments depend on the context of preceding interface events in order to guarantee plausible component interaction.

Security objectives avoiding the two types of availability attacks are enforced by the policy patterns listed below:

- **Denial-of-service minimum waiting time**: An interface event may only be executed if a minimum waiting time period elapsed since a similar event was executed.

- **Denial-of-service enabling history**: The enabling condition of an interface event depends on the context of certain preceding interface events and additionally on a minimum waiting time period.
- **Blocking maximum waiting time**: An interface event has to be executed before a maximum waiting time period expired since a certain preceding interface event.
- **Blocking enabling history**: According to the context of certain preceding interface events an interface event has to be executed before a maximum waiting time period expired.

In order to support non-repudiation we use the following pattern:

- **Event logging**: A component has to log an executed or received event together with a unique signature with a trusted third party logging service.

All of these security policies can be modeled formally as state transition systems. The policy enforcement wrappers (cf. Sec. 3) implement corresponding state machines by means of state representations and state change operations (Herrmann and Krumm, 2001). During run-time, the occurrences of component interface events trigger state changes. Thus, the state machines keep track of the execution history and the compliance of interface events with the security policies can be checked.

For the specification of the policies, we used the formal specification technique cTLA (cf. Herrmann and Krumm, 2000a) which facilitates specifications of safety, liveness, and real-time (Herrmann and Krumm, 2000b) properties in a process style similar to high-level programming languages. Furthermore, cTLA supports the design of constraint-oriented specifications (cf. Vissers et al., 1988). Thus, different security policies may be specified and observed separately. Moreover, we like to mention that cTLA can additionally be used for verifying that the combination of the implemented policy enforcements of an application guarantees abstract application security properties.

We propose, that component vendors add cTLA specifications of suitable security policies to component contracts. Before incorporating a component into an application, an application developer may check and extend the proposed security policies. The detailed policy specifications used in the following e-requisitioner example can be accessed via WWW (URL: ls4-www.informatik.uni-dortmund.de/RVS/P-SACS/eReq). The transformation of these cTLA specifications to implementing Java code has been performed manually since a suitable cTLA-to-Java compiler is not yet operational.

5. Component-Structured E-Procurement Application

To support standardization of electronic procurement (e-procurement) procedures, the OBI consortium issued a set of specifications for Open Buying on the Internet (OBI, 1999). According to these specifications the architecture for e-procurement activities consists of a buying organization, a selling organization, a payment authority, and a requisitioner. In behalf of the buying organization the requisitioner carries out orders of goods at the selling organization which are paid by means of the payment authority. The corresponding OBI business-to-business (B2B) model consists of seven successive steps:

1 The requisitioner asks the buying organization for hyperlinks to merchant servers of selling organizations.

2 The requisitioner requests the selling organization to offer tenders for the desired goods.

3 Each selling organization creates a tender and maps it into an OBI order request which is compatible to the EDI standard (DISA, 2001) and transfers it to the buying organization either via the requisitioner or directly.

4 Based on the tenders, the requisitioner and possibly other entities of the buying organization select a winning selling organization and generate an order.

5 The completed order is formatted as an EDI-compatible OBI order object and is transferred to the winning selling organization.

6 The selling organization fulfills the order.

7 In behalf of the selling organization the payment authority issues an invoice to the buying organization and receives a payment.

Our example system performs the commodity management of fast-food franchise restaurants. It was developed on the basis of the Sales-Point-Framework (Schmitz, 1999). This framework is non-profit and facilitates the construction of various shop systems. It supports business functions like buying, selling, or leasing goods as well as administrative functions like accounting, storekeeping, and management of product catalogs. The framework is implemented in Java but was originally not component-structured. Therefore we adapted the commodity management system and created three Java Bean-based components which realize the sale functions of the restaurant, the management of the counting stock, and the catalog of offered products.

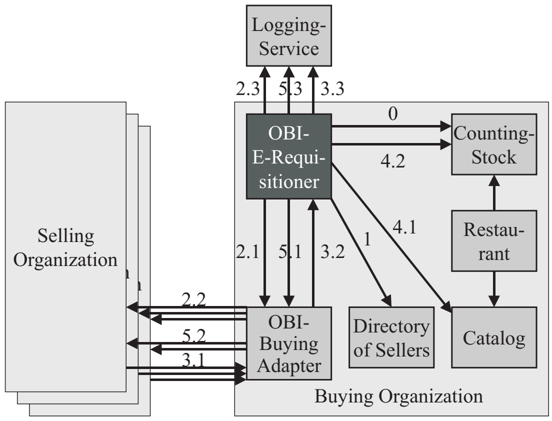

Figure 2. E-Procurement System

To enable automated e-procurement of the food and beverages, we created and added three other components making the restaurant to an OBI buying organization (cf. Fig. 2). First we extended the OBI specification in order to integrate automated procurement which in contrast to OBI is not performed by humans but by an electronic requisitioner component *OBI-E-Requisitioner*. Moreover, we added a *Directory of Sellers* containing the addresses and range of goods for sale of the selling organizations. Finally, the *OBI-Buying Adapter* manages the formatting of tender requests, tenders, and orders according to the OBI specification and acts as an interface to the selling organization. The composition of the six components realizes the buying system. Moreover, a group of selling systems was developed based on the SalesPoint-Framework. Finally, we created a trusted third party logging service in order to support non-repudiation of transactions. Since we are mainly interested in the order process, we omitted the payment authority for the sake of simplicity. The components can be downloaded from the WWW-project page (URL: ls4-www.informatik.uni-dortmund.de/RVS/P-SACS/eReq).

Due to the integration of automated procurement we had to extend OBI by format definitions for machine-processable tender request messages. Since the EDI standard (DISA, 2001) does not support tender requests, we encoded tender requests, tenders, and orders in commercial eXtensible Markup Language (cXML, 2001). This more modern B2B encoding standard is based on the popular XML and supports not only purchase orders and tenders but also tender requests.

The realization of the procurement steps is delineated by the edge labels in Fig. 2. Since the e-requisitioner manages not only the procurement process but also the decision, when to order, the procurement starts with a new step 0. Here, the e-requisitioner inspects the counting stock in intervals. If new goods are needed, it requests the addresses of selling organizations from the directory of sellers (step 1). Thereafter tender requests are generated and forwarded to the selling organizations via the buying adapter (step 2). The sellers react with tenders which are sent to the buying adapter and delivered to the e-requisitioner (step 3). In step 4 the e-requisitioner consults the catalog and the counting stock beans, makes a procurement decision based on the tenders, the stock volume, and the sale prices, and creates the order objects. Finally, the orders are sent to the buying adapter which forwards them to the corresponding selling organizations (step 5). The steps 6 and 7 realizing the payment are omitted. In order to log the tender requests, incoming tenders, and orders, in the steps 2, 3, and 5 the requisitioner sends the corresponding log data to the logging service.

6. Component Behavior Enforcement

Since the procurement process is controlled by the electronic requisitioner, correct and secure execution of this component is crucial for the whole application. Malicious behavior of the *OBI-E-Requisitioner* may lead to various security violations like forwarding of competitor's tenders to preferred selling organizations, ordering not from the least expensive seller, hurting the buyer by too large, too small, resp. too late orders, or repudiation of orders. Assuming that the e-requisitioner was procured from a possibly not trustworthy company, we applied the security patterns described in Sec. 4. Under instantiation of these patterns, a set of suitable policies were designed and described by cTLA specifications. The cTLA specifications were transformed to Java code which was integrated into the wrapper of the e-requisitioner component.

Following confidentiality protecting policies are used:

- A tender request contains only articles which, according to the directory of sellers, are in the range of articles offered by the particular seller (Data flow access).
- A tender is requested only from sellers contained in the directory of sellers (Data flow access).
- The order amount for an article depends unambiguously from the amount of the particular article in the stock (Hidden channel functional dependency).
- A tender request and an order may be executed only if the last order was carried out in the meantime (Hidden channel enabling history).

The first two policies guarantee that information about the portfolio of the buying organization and the existence of the procurement procedure are only forwarded to appropriate sellers. The two other security policies make the use of hidden channels (e.g., for information about competitors' tenders) more difficult by avoiding non-deterministic interface behavior.

With respect to integrity following policies are used:

- An order may be generated only after a certain minimum number of tenders were received (Integrity enabling history).
- The requisitioner orders one of the least expensive tenders (Integrity enabling history).
- The values in the counting stock, the catalog, the directory of sellers, the *OBI-Buying Adapter*, and the logging service are not changed (Integrity enabling condition “false” for modifying operations).
- The amount ordered is in an interval between a certain minimum and maximum (Integrity enabling condition).

The first two security policies guarantee that all selling organizations have a fair chance to win an order. Attacks against other components are avoided by the third objective while the last one prevents orders with unreasonable amounts of an article.

To prevent attacks against the availability of the system, the security policies below are used:

- Operations of the counting stock, the catalog, the directory of sellers, and the buying adapter are called only after minimum waiting time intervals (Denial-of-service minimum waiting time).
- The counting stock is polled within maximum waiting time intervals (Blocking maximum waiting time).
- If, according to the counting stock, the number of a certain article is low, a procurement process for this article is started within a maximum time interval (Blocking enabling history).
- After receiving the threshold number of tenders an order is executed within a maximum waiting time (Blocking enabling history).

Denial-of-service attacks against the partner components are prevented by the first security objective while the other specifications guarantee that orders are executed timely to avoid cleared stocks.

Finally, to assure that the buying organization can audit requested and incoming tenders as well as all orders, the following non-repudiation security policy is used:

- Tender requests, tender deliveries, and orders are logged at the logging service (Event logging).

7. Concluding Remarks

We reported on the use of formal security policy specifications in component contracts. The compliance of the real behavior with the contracts is enforced at run-time by means of wrappers. In our e-procurement example the run-time enforcement causes a performance penalty between 5 and 10%. While this penalty seems acceptable, it can be reduced by using a trust manager which controls the amount of observation according to the current trust value, which is assigned to the component by a trust information service (cf. Herrmann, 2001). Current work concentrates on the development of a component system security framework which — besides of enforceable policies — additionally provides patterns for the specification of abstract application security properties. Moreover, the framework contains proof theorems facilitating the formal proof that the security properties of an application are fulfilled by its components.

References

Avolio, F. M. and Ranum, M. J. (1994). A Network Perimeter with Secure External Access. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, Glenwood.

Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. (1995). Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–284. ACM.

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *IEEE Computer*, 32(7):38–45.

Biskup, J. and Eckert, C. (1994). About the enforcement of state dependent security specifications. In Keefe, T. and Landwehr, C., editors, *Database Security*, pages 3–17. Elsevier Science (NorthHolland).

cXML (2001). *cXML User's Guide*. cXML.org, 1.2.006 edition.

DISA (2001). *X12 Standard*. Data Interchange Standards Association, release 4050 edition.

Ferrari, E., Samarati, P., Bertino, E., and Jajodia, S. (1997). Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140, Oakland.

Fraser, T., Badger, L., and Feldman, M. (1999). Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. (1996). A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*.

Herrmann, P. (2001). Trust-Based Procurement Support for Software Components. In *Proceedings of the 4th International Conference on Electronic Commerce Research (ICECR-4)*, pages 505–514, Dallas. ATSMA, IFIP.

Herrmann, P. and Krumm, H. (2000a). A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337.

Herrmann, P. and Krumm, H. (2000b). A Framework for the Hazard Analysis of Chemical Plants. In *Proceedings of the 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD2000)*, pages 35–41, Anchorage. IEEE CSS, Omnipress.

Herrmann, P. and Krumm, H. (2001). Trust-adapted enforcement of security policies in distributed component-structured applications. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 2–8, Hammamet. IEEE Computer Society Press.

Jøsang, A. and Knapskog, S. J. (1998). A metric for trusted systems. In *Proceedings of the 21st National Security Conference*. NSA.

Khan, K., Han, J., and Zheng, Y. (2001). A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components. In *Proceedings of the Australian Software Engineering Conference (ASWEC'01)*, pages 117–126, Canberra. IEEE Computer Society Press.

Kozen, D. (1998). Efficient code certification. Technical Report 98–1661, Computer Science Department, Cornell University.

Kozen, D. (1999). Language-Based Security. In Kutylowski, M., Pacholski, L., and Wierzbicki, T., editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS'99)*, Lecture Notes in Computer Science 1672, pages 284–298. Springer-Verlag.

Mallek, A. (2000). Sicherheit komponentenstrukturierter verteilter Systeme: Vertrauensabhängige Komponentenüberwachung (in German). Diploma Thesis, Universität Dortmund, Informatik IV, D-44221 Dortmund.

Monroe, M. A. (1993). Security Tool Review: TCP Wrappers. *;login:*, 18(6):15–16.

Morrisett, G., Walker, D., Crary, K., and Glew, N. (1998). From System F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego.

Myers, A. C. and Liskov, B. (1998). Complete, Safe Information with Decentralized Labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 186–197, Oakland.

Necula, G. C. (1998). *Compiling with proofs*. PhD thesis, Carnegie Mellon University.

OBI (1999). *OBI Technical Specifications — Open Buying on the Internet*. OBI Consortium, draft release v2.1 edition.

Schmitz, L. (1999). The SalesPoint Framework — Technical Overview. WWW: ist.unibw-muenchen.de/Lectures/SalesPoint/overview/english/TechDoc.htm.

Schneider, F. B. (1997). Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, Lecture Notes in Computer Science 1320, pages 1–14. ACM SIGPLAN, Springer-Verlag.

Szyperski, C. (1997). *Component Software — Beyond Object Oriented Programming*. Addison-Wesley Longman.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. (1996). TIL: A type-directed optimizing compiler for ML. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM SIGPLAN.

Vissers, C. A., Scollo, G., and van Sinderen, M. (1988). Architecture and specification style in formal descriptions of distributed systems. In Agarwal, S. and Sabnani, K., editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 189–204, Elsevier. IFIP.

Wabbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. (1993). Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 203–216. ACM.

Zöllner, J., Federrath, H., Klimant, H., Pfitzmann, A., Piotraschke, R., Westfeld, A., Wicke, G., and Wolf, G. (1998). Modeling the security of steganographic systems. In *Proceedings of the 2nd Workshop of Information Hiding*, LNCS 1525, pages 345–355, Portland. Springer-Verlag.