Wrappers — A Mechanism to Support State-based Authorisation in Web Applications

Martin S Olivier

Computer Science, Rand Afrikaans University, PO Box 524, Auckland Park, 2006, South Africa; e-mail: molivier@rkw.rau.ac.za

Ehud Gudes

Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel; e-mail: ehud@cs.bgu.ac.il

Abstract

The premises of this paper are 1) security is application dependent because application semantics directly influence proper protection; but 2) applications are generally too complex to be trusted to implement security as specified by the given security policy. These problems are aggravated if the application operates over time and space.

This paper proposes the use of a simple program (a "wrapper") that has enough knowledge about a specific application's potential states and the actions that are permissible in each state. Using this knowledge, it is able to filter requests that should not reach an application at a given point.

Key words: Web Security, Application security, Access control, Wrappers, State-based authorisation.

1 Introduction

It is generally accepted that security should ideally be based on the concerned application: Only when the application logic is considered, is it possible to precisely determine security-related concerns such as the need-to-know or context sensitive security requirements. Such requirements have most notably been expressed in the workflow context where, for example, someone's authorisation to modify a given document depends on the current state of the workflow process.

Such context-dependent security measures are clearly not restricted to workflow systems. Applications in general exhibit this requirement. As a simple example note that in almost any sensitive application it should not be possible to do any processing before proper user authentication has been performed. Although such a simple requirement seems obvious, it is a generally accepted fact that many complex applications do contain backdoors that enable the developer access to the system without going through the normal authentication procedure [10]. Many forms of attack will also deviate from normal application flow of logic. One well-known attack is to overflow buffers on the stack and so take control of the system. This form of attack was, amongst others, used in the Internet worm [10] and was again recently exposed as vulnerabilities in well-known Microsoft and Netscape products [4].

It is our contention that application-layer security should be part of the design process of any major application and that an appropriate security model should be implemented as part of the application development process. It is, however, a fact that real-world applications are complex by nature and that it is very hard to ensure their correctness in general, and the implementation of their security features in particular. Add to this the fact that application designers are often not security experts and that weak methods are often employed to protect the system (in addition to the fact that the system is complex to protect), and concern is indeed warranted.

The option that interests us in the current paper is the possibility to place a 'wrapper' around the application such that the user interface falls outside the wrapper and all communication between the user interface and the application has to occur via this wrapper. In some senses this wrapper is similar to a firewall that protects the internal network of an organisation from the untrusted external network. The wrapper is, however, fundamentally different from a firewall since it is intended to wrap a specific application, possibly even on a single host without any notion of networking.

Such a wrapper is obviously a much smaller system since it does not implement the application logic, but simply forms an additional security layer. Since the wrapper is much simpler, it is much easier to verify that it does indeed implement the required security policies and therefore much easier to trust. In addition, since the primary concern of the wrapper is security, it is natural that it may be developed by security experts rather than application experts that may mean that common pitfalls will be avoided.

Although not the only (or necessarily the best) way to separate the application from its user interface, we shall use a Web browser as the user interface component and HTTP as the communication protocol.

This paper is structured as follows. Section 2 contains background material

about HTTP. Section 3 gives an overview of the approach that we present. Section 4 discusses implementation issues associated with our approach. Section 5 compares our approach to existing approaches while section 6 concludes the paper.

2 Background

This paper assumes knowledge of the HTTP protocol. We therefore give a very brief introduction to this protocol. Readers interested in a more detailed treatment are referred to RFC 2616 [9].

The HTTP protocol is used to connect a client (typically a Web browser) to a Web server. It establishes this connection using the TCP/IP protocol suite.

The primary request sent by a browser is a `GET` request. It may be formulated in a number of ways but the following is typical when programs are to be executed:

```
GET /cgi-bin/program.cgi?param1=val1&param2=val2 HTTP/1.1
HOST: server.domain
```

The requested program is then executed and its output is sent to the browser. The program has a standard mechanism to access the values of parameters sent to it (such as `param1` and `param2` in the example above). The output of the program is typically encoded in HTML and rendered by the browser.

HTTP is a stateless protocol where each request sent from the browser is replied to by the server and then 'forgotten' — ie it does not influence further operation of the protocol. Two mechanisms are typically used to add state information when required. Firstly, cookies may be used. Cookies are short files written to the disk of the client machine; the values of these files can be obtained during subsequent requests and used to establish the current state of a longer interaction. The second mechanism that is commonly used is a session identifier. When a 'session' is initiated, the server sends some unique identifier along with its response such that the identifier will be sent back to it with a subsequent request. This identifier is then passed to and fro between the client and server and used to keep track of the session state.

3 State-based security checking

The application wrapper proposed in this paper is intended to form a layer between the application and the user interface such that the wrapper contains enough information to verify state-based access controls. However, to be useful,

it is essential that the wrapper (1) is simple enough that it potentially has a much higher level of trust than the application; and (2) does not have access to any application data, so that if the wrapper is somehow compromised, the attacker has gained very little.

The first goal will be reached by showing that it is possible to produce a generic wrapper that may be configured for any application. Furthermore, it will be shown that this can indeed be done without giving this wrapper access to sensitive information.

To accomplish this consider the data that is required by the wrapper. We will assume that the user interface is a thin client that contains no application specific logic. Requests are passed from this client, through the wrapper to the application, which is implemented as a server. Responses are returned along the same route in the opposite direction.

3.1 Wrappers and Firewalls

Note that wrappers are similar in many ways to application gateway firewalls. Wrappers are, however, (1) intended for a more general class of applications than the services to which application gateways have traditionally been applied, and (2) wrappers are not intended as a defence between an organisation's network and the outside world, but all accesses to a wrapped application are forced to go via its wrapper — even from the most trusted machines on a local network.

3.2 Basic access control

In order to keep the wrapper as simple as possible, we will assume that it is not concerned with authentication: The user is expected to obtain a 'ticket' from an authentication server and present it with every request relayed by the wrapper. For simplicity we will assume that roles are used. Therefore, it is assumed that every request will consist of a (certified) role r and the action q to be performed. How the role will be 'certified' is not important for the discussion of the concept and an explanation of such certification will be delayed until section 4. For the current discussion it is only necessary to assume that an unauthorised user (ie someone who is not authorised to operate in role r) will not be able to obtain such a certificate.

If the wrapper has access to a table that contains all valid (role, action) pairs, it is simple for the wrapper to enforce the required checking.

3.3 The single-session case

More challenging than simple access control (and more relevant to the current paper) is enforcement of state-based access control. State-based access controls are useful whenever a partial ordering of actions exist in which they have to be performed. (Consider the example where a customer is expected to supply credit card details *before* being allowed to download software purchased online.)

The set of requests may be purely sequential or may represent a complex partially ordered set of requests such as is customary in workflow applications (see for example [1]).

The essential question that the wrapper needs to be able to answer is, is some request t valid if it follows a series of requests $t_1, t_2, t_3, \ldots, t_n$? This is, amongst others, a well-known problem addressed by formal languages: given two sentential forms T_1 and T_2 is it possible to derive T_2 from T_1, ie $T_1 \Rightarrow T_2$? Using a grammar for state-based authorisation has been dicussed in detail by Biskup and Eckert [2]. That paper also describes the translation of such specifications to finite automata, that are used by *monitors* to enforce the specifications.

It is obvious that most existing online applications have a very simple state-based security requirement, such as login must precede any further interaction with the application and payment must precede download of purchased items (assuming an online shop that sells 'soft' goods). The grammar based approach has the potential to specify policies such as allow no further interaction from a customer during the current session who has three times specified credit card details that were not accepted by the bank; this may, for example be specified using a grammar such as

$$\begin{array}{lcll} \Sigma & \rightarrow & lXPY & \text{where} \\ P & \rightarrow & c|c'c|c'c'c|c'c'c'F & \end{array} \begin{cases} l \text{ is the login request} \\ X \text{ is any pre-payment request} \\ Y \text{ is the downloading of purchased item} \\ P \text{ is the payment request} \\ c \text{ is an accepted credit card specification} \\ c' \text{ is a denied credit card specification, and} \\ F \text{ is any request that is still allowed after} \\ \text{failing credit card verification, such as logoff} \end{cases}$$

Exactly how a wrapper knows that credit card verification has failed will be discussed below. It is clear from the specification above that 'payment' only succeeds if credit card verification occurs within three attempts.

The example above illustrates a crucial requirement of wrappers: it is essential to keep the wrapper synchronised with the application. Since the wrapper is explicitly barred from accessing application data, a wrapper may allow an operation that is denied by the application. Therefore the wrapper needs to be able to monitor responses from the application to the client. The wrapper now proceeds as depicted in figure 1.

```
while (true)
    acceptrequest(ρ_i) with ρ_i = (r_i, q_i)
    if q_i is not acceptable in current state or r_i is not allowed to request
    q_i then
        send 'reject' to client
    else
        send ρ_i to application
        receive response a from application
        if a.success then
            append q_i to log // state successfully exited
            enter new state based on q_i
        else
            append q'_i to log // alternative state used
            if q'_i is not acceptable in current state then
                send 'reject' to client; send 'abort' to application
            else
                enter new state based on q'_i
```

Fig. 1. Pseudocode for wrapper

3.4 The multi-session case

Thus far we have only considered 'transactions' that occur within a single session. The ultimate challenge is to handle 'long transactions' that span multiple sessions (and that are potentially executed by more than one user). This scenario is typical of workflow applications where one user submits a claim during one session. During a subsequent session an approver works sequentially through submitted claims and either approves or denies them. Finally, during a third session, the cashier looks at approved claims and issues cheques.

In order to handle long transactions it becomes necessary to associate a transaction identifier t with each long transaction. It is obviously necessary to set and modify t when required: consider the approver who looks at one transaction after the other — each time a different transaction is involved, and whether the transaction is a candidate for approval depends on the state of the particular transaction. Obtaining t may be accomplished by identifying all requests (to the wrapper) which may lead to an updated value of t; t may be extracted from a parameter that accompanies a request and/or determined from the response $a.transaction$ that the algorithm above is now expected to

return.

An interesting point to note is that a user's behaviour may now be governed by two distinct state-oriented policies. To illustrate, consider the familiar claim example. When the client logs onto the claim system, the system may assign a transaction identifier to this specific claim. From this point on the actions of the client are governed by the rules for claim submission. Suppose that the client completes submission of the claim, is the client now permitted to look at the status of another claim during the same session? This is clearly a session-state issue rather than a transaction-state issue.

We propose the following solution. Firstly, rules for transactions are specified. For the claim example this may involve specifying the sequence for submitting and subsequently approving the claim, until it is eventually settled. Different 'sessions' may then be identified. For this example a 'client session' may, for example, be specified as a session initiated by a client login request, which may then be followed by a request to initiate submission or a status request. The submission or status request is governed by the rules for the transaction, rather than the session. Consider

$$\begin{array}{rcl} \Sigma & \rightarrow & m_c T_c \\ T_c & \rightarrow & c \oplus \mid s \oplus \end{array} \quad \text{where} \begin{cases} m_c \text{ is a client menu request} \\ T_c \text{ is a client transaction} \\ c \text{ is the first step of a claim submission} \\ s \text{ is the first step of a status query; and} \\ \oplus \text{ is discussed below} \end{cases}$$

$\oplus$ is used in the specification above to indicate that the preceding request (c or s) is the start of a sequence that will be governed by a transaction specification. This implies that a transaction has to be identified by c and s.

Note that it is simple to modify the session specification to $\Sigma \rightarrow m_c T_c^*$ or even $\Sigma \rightarrow (m_c T_c)^*$ to express different policies.

At this point it becomes necessary to consider ordinary transactions. Consider an on-line banking application where a client selects an account from which money is to be transferred with one request and an account to which the money should be transferred with a subsequent request. If the session fails at this point, it is unlikely that the application will allow the client to continue from this point at a (significantly) later stage: The basic actions are likely to be grouped into a transaction that are to be executed as a unit. The question is whether the wrapper needs to be aware of such transactions. We argue that it is not the case, as long as the wrapper knows where a new session is allowed to start: In the case of the banking example, a session is allowed to start at the first account selection and nowhere else. We foresee that session specifications will often be used merely to indicate such possible starting points.

Note that the rules for transaction state access control will typically not be provided as a grammar by the security officer, but may be automatically derived from the workflow specification.

4 Implementation

The use of the Web as a means to implement various applications has increased tremendously over the last number of years. In addition, the separation between user interface (in a browser) and an application server (as required by our approach) is indeed present in the Web environment.

4.1 Filtering messages

It is relatively simple to write a wrapper that relays (filters) requests between the client and application as required for wrappers.

To illustrate the operation of a wrapper in the Web environment, consider the claim example again. The specification in figure 2 is not intended as an illustration of a specification language, but rather to make some of the issues previously discussed more concrete.

Lines beginning with a hash (#) are intended as comments. The initial section of the configuration specifies all messages (or requests), along with the parameters that are employed. The *role* parameter is implicit in all messages and not shown. It will be discussed below.

When a parameter is marked with an asterisk (such as `transaction*` with `view-a-claim`), it means that this value is not directly provided by the user, but is typically retrieved from a response to a previous message. In the example case `view-a-claim` is preceded by a `view-claims` message. The intention is that the response to the `view-claims` message is a list of claims ready for approval, together with their associated (long) transaction identifiers. Therefore, when the user clicks on a particular claim, the form is composed so that the 'selected' transaction identifier is sent as a parameter with the `view-a-claim` message.

We assume that the selected names of messages are clear enough for the reader to interpret. However, keep in mind that the names refer to messages, not screens. Therefore an approver may look at a screen of claims ready for approval; when the approver clicks on one such claim, the `view-a-claim` request is sent and this is followed by a screen with claim details. On this screen the approver may click on approve or deny

```
MESSAGES
# Messages for claiming
  client-welcome         ()
  view-form              (session, transaction*)
  submit-claim           (session, transaction, incident, amount)
# Messages for approval
  admin-welcome          ()
  view-claims            (session)
  view-a-claim           (session, transaction*)
  approve-claim          (session, transaction)
  deny-claim             (session, transaction)
# Messages for payment
  view-approved-claims   (session)
  view-an-approved-claim (session, transaction*)
  issue-cheque           (session, transaction)

TRANSACTIONS
             -> Submission Approval Payment
  Submission -> view-form submit-claim
  Approval   -> view-a-claim { approve-claim | deny-claim end }
  Payment    -> view-an-approved-claim issue cheque end

SESSIONS
# Claim session
    -> client-welcome view-form...
# Approval session
    -> admin-welcome
         { view-claims view-a-claim... }*
# Payment session
    -> admin-welcome { view-approved-claims view-an-approved-claim... }*
```

Fig. 2. Illustrative wrapper specification

The transaction rules should also be self-explanatory for this simple example. Note that this specification is rather inflexible and that more flexibility will be required for real-world applications. We contend that our approach is capable of handling more flexible requirements but do not illustrate it in the current paper for lack of space.

In the case of specific sessions, ellipses have been used to indicate that a portion of a session will be controlled according to a transaction specification rather than a session specification. (In the previous section we used $\oplus$ for this, but this symbol is not available on keyboards for typical configuration files.)

It is now clearly simple for the wrapper to apply the algorithm given in the previous section. When a message arrives (1) ensure that the message may indeed be sent by a user as stated in the role parameter (see below for a discussion of this parameter); (2) ensure that the message is valid in the current

session state, if the message forms part of a session state specification; and (3) ensure that the message is valid in the current transaction state, if the message forms part of a transaction state specification.

Transaction state needs additional consideration. In the single session/single transaction case this is not a problem: the wrapper can maintain the state in memory, and this way can easily check whether the next (role, action) pair is valid or not.

In the multi session/multi-transaction case this is more complicated. One option is for the wrapper to maintain all these states in a local database. This is however too complex since it defeats the wrapper simplicity principle, and also takes upon itself much application functionality. This is even more complex when the multiple sessions are initiated in different sites and involve different wrappers. The solution we advocate is the following. Using a specification similar to the one discussed in section 3.4, the wrapper can know exactly which (role, action) pairs are valid at the beginning of a session. The suitability of such a pair to the specific *transaction state* will be tested by the application itself. If the answer will be positive the wrapper can start to maintain the state in memory so long as this transaction is active. Thus no local database and no application database is required by the wrapper.

4.2 Role identifiers

The role parameter is different from other parameters since authorisation is based on it. The following approach solves the major problems: Firstly, the user requests a role identifier from an authentication server. Note that the authentication server is independent of wrappers. When requesting the identifier, the user states the requested role and provides the required credentials, such as a user name and password. The authentication server then compiles a certificate containing role information such as the name of the role and its period of validity and then signs this information with its (the authentication server's) private key. If we assume that an encrypted channel (such as SSL) is used between the user and the authentication server, this certificate cannot be misappropriated by an eavesdropper. A similar argument shows that it cannot be misappropriated between the user and the wrapper or the wrapper and the application if encryption is used. It is obviously simple for the wrapper (and the application) to verify the authenticity of the role identifier by using the authentication server's public key. Note that this approach presents a simplified version of *secure cookies* as presented by Park *et al* [14], but avoids the use of cookies.

4.3 Wrapper and Application Relationship

A major issue in the proposed scheme is the extraction of the state based behaviour from the application in order to specify it precisely for the wrapper. If one uses an external tool for such specification one runs the danger of creating inconsistencies between the application and the wrapper, and maintaining the wrapper specifications in case the application changes. An automatic or semi-automatic tool is much more desirable.

Let us assume that the application consists of a set of CGI scripts. There are several alternatives to generate the state-role/action sequences. Usually, such CGI scripts are very simple and as was explained above retrieve commands sent to them using GET or POST using some explicit mechanism. The semi-automatic scheme we propose involves the insertion — *by the application developer* — of statements which specify the desired role for each such GET/POST. In addition the application developer can insert statements that assert that some condition has been met (eg `assert user_logged_on`) in some scripts where this condition has indeed been met and state that the condition is required to execute other scripts (eg `require user_logged_on`). (Such statements may be handled by a pre-processor.) Then a simple program-flow analyzer can generate all the possible sequences of action/role pairs. Later on the application developer will need to maintain only such CGI scripts.

Another possibility is to write a simple parser which will scan the CGI scripts and for each GET/POST it finds will inquire the application developer for the relevant role/action pair. Then the generation of possible sequences will be done as mentioned before. Yet, another possibility is to develop a high-level tool for specifying states and actions (similar to State-charts [11] or Petri-nets[1]) and from that tool to *automatically* generate both the state-based sequences and skeletons for the CGI scripts required by the application.

5 Comparison with other work

The Wrappers idea is related to several other works which appeared in recent years. The idea of extracting security policies from an application and enforce and maintain them separately has appeared before. In [12] it is argued that in a distributed object environment, one cannot leave the security enforcement to monolithic components such as DBMSs. Since, in such systems, requests usually go through mediators or brokers like CORBA or DCOM we should associate security enforcement with these mediators. [12] is very general and does not go into the details of what kind of authorisation should be handled by the brokers and what should remain with the application or DBMS.

Our work differs from that of Biskup and Eckert cited earlier [2] since (1) a greater emphasis is placed on isolation of the wrapper; (2) transactions where multiple subjects cooperate are considered; and (3) it is set in the Web context.

Another area where State-based security was proposed is in the problem of protecting *mobile agents*. For example, in [7] there is a state appraisal mechanism which associates a mobile agent with a state appraisal function. When a roaming agent reaches a new execution environment, the appraisal function is evaluated passing as a parameter the agent's current state. In our case both the application and the wrapper are not mobile, but the need to use state information is similar.

The DOK system for federated databases proposed by Tari [16] has a complex structure of agents enforcing security on behalf of the individual application or local DBMS (he even uses the term "Wrapper" but for translating global to local requests only). Our wrapper on the other hand is quite simple but has a focused goal — providing state-based security for Web-based applications. Therefore, issues such as translating queries are not handled by it.

The TIHI project by Wiederhold *et al* [17] is quite close to our ideas. It uses an object called a "Security Mediator" for enforcing the security policies of a Web-based medical application. It is also implemented using CGI scripts. It is, however, application specific, and not generic like our wrapper. It also handles all authorisation and not only the state-based.

Another paper on Role-based security by Demurjian *et al* [6] is also related to our work. They suggest the concept of an OSA (Object Security Officer) which separates an object from the outside world like a firewall. However their OSA is linked to the application object much tighter than our wrapper, since it invokes the object methods directly.

Finally, in our own work on workflow security [8,13] we showed how we can specify and enforce history and dynamic authorisation rules. All the authorization is done by the workflow security administrator object. Thus it is tightly coupled with the workflow specification. Again, the Wrapper idea here is not as tightly coupled with the application. It must be synchronised with it with respect to the major states and roles but it still leaves data dependent and dynamic authorization checks to the application.

6 Conclusion

This paper has presented an approach to reinforce application security in an environment such as the Web by introducing another layer of defence between

the firewall and the application itself. It is important to remember that this layer is not intended to remove any responsibility from either the firewall or the application, but rather to provide additional security.

It has been demonstrated that the concept is indeed simple — a requirement for trust — and can be based on a simple configuration approach. The paper has not investigated suitable specification approaches in any detail but merely posited that it should be possible to either specify the wrapper's behaviour manually for a simple application (such as many current online stores) or to create a tool that will automatically derive a specification from others such as those used for workflow systems in general. This remains to be verified.

Further, exactly how the specification is converted to a wrapper has not been considered in detail in this paper. Much work has been done over many years to generate parsers for arbitrary grammars and this should therefore be simple.

References

[1] V. Atluri and W.K. Huang, An extended petri net model for supporting workflow in a multilevel secure environment, in: P. Samarati and R.S. Sandhu, eds., *Database Security X: Status and Prospects* (Chapman & Hall, London, 1997), 240–258.

[2] J. Biskup and C. Eckert, About the Enforcement of State Dependent Security Specifications, in: T.F. Keefe and C.E. Landwehr, eds., *Database Security VII: Status and Prospects* (Elsevier, 1994), 3–17.

[3] F. Casati, S. Ceri, B. Pernici, G. Pozz, Conceptual Modelling of Workflows, *Proc. of the Object-oriented and Entity-Relationship Conf.*, Australia, 1995.

[4] CERT, *Buffer Overflow in MIME-aware Mail and News Clients*, CERT Advisory CA-98.10, 1998.

[5] W. Ford and M.S. Baum, *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption* (Prentice Hall, Englewood Cliffs, 1997).

[6] S.A. Demurjian, Y. He, T.C. Ting and M. Saba, Software Agents for Role Based Security, in: V. Atluri and J. Hale, eds., *Research Advances in Database and Information Systems Security* (Kluwer, Norwell, 2000), 79–93.

[7] W.M. Farmer, J.D. Guttman, and V. Swurp, Security for Mobile Agents: Authentication and State Appraisal, *4th European Symp. on Research in Computer Sceurity*, LNCS Vol 1146, 1997, 118-130.

[8] E. Gudes, M.S. Olivier and R.P. van de Riet, Modelling, Specifying and implementing workflow security in Cyberspace, *Journal of Computer Security*, **7**, 4 (1999), 287–315.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Internet Society, 1999).

[10] S. Garfinkel and G. Spafford, *Practical Unix & Internet Security, 2nd. ed.* (O'Reilly, Sebastopol, 1996).

[11] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: the STATEMATE Approach* (McGraw-Hill, 1998).

[12] C.D. McCollum, D.B. Faatz, W.R. Herndon, E.J. Sebes and R.K. Thomas, Distributed object technologies databases and security, in: T.Y. Lin and S. Qian, eds., *Database Security XI: Status and Prospects* (Chapman & Hall, London, 1998), 17–31.

[13] M.S. Olivier, R.P. van de Riet and E. Gudes Specifying Application-level Security in Workflow Systems, in: R. Wagner, ed., *Proceedings of the Ninth International Workshop on Security of Data Intensive Applications (DEXA 98)* (IEEE, Los Alamitos, 1998) 346–351.

[14] J. Park, R. Sandhu and S. Ghanta, RBAC on the Web by secure cookies, in: V. Atluri and J. Hale, eds., *Research Advances in Database and Information Systems Security* (Kluwer, Norwell, 2000), 49–62.

[15] L.D. Stein, *Web Security: A Step-by-Step Reference Guide* (Addison-Wesley, Reading, 1998).

[16] Z. Tari, Designing security agents for the DOK federated system, in: T.Y. Lin and S. Qian, eds., *Database Security XI: Status and Prospects* (Chapman & Hall, London, 1998), 35–59.

[17] G. Wiederhold, M. Billelo and C. Donahue, Web implementation of a security mediator for medical databases, in: T.Y. Lin and S. Qian, eds., *Database Security XI: Status and Prospects* (Chapman & Hall, London, 1998), 60–67.