Securing Service-Oriented Systems Using State-Based XML Firewall*

Abhinay Reddyreddy and Haiping Xu
Computer and Information Science Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
{g_areddyreddy, hxu}@umassd.edu

Abstract. *Web services security has been a challenging issue in recent years because current security mechanisms, such as conventional firewalls, are not sufficient for protecting service-oriented systems from XML-based attacks. In order to provide effective security mechanisms for service-oriented systems, XML firewalls were recently introduced as an extension to conventional firewalls for web services security. In this paper, we present a state-based XML firewall architecture that supports role-based access control and detection of XML-based attacks. We develop a detailed design of our state-based XML firewall by defining state-based information, user information, and various access control policies and detection rules. The detection rules are modularized into separate units, which support real-time detection and verification of various types of XML-based attacks using state-based information and user information. To illustrate the effectiveness of our approach, we develop a prototype state-based XML firewall, and demonstrate how XML-based attacks can be efficiently detected.*

1. Introduction

Enterprises are increasingly employing web services technology in order to achieve interoperability at application levels. Web services are both platform and language independent service components that can be exposed using a standard Web Services Description Language (WSDL) and registered at UDDI registries. They can be automatically discovered over the Internet by potential clients, and support loosely coupled interactions between applications through a standard XML-based protocol, called Simple Object Access Protocol (SOAP). Since web services create open interfaces into core enterprise applications and data, attacks on web services can be more severe than those attacks perpetrated via e-mail, web servers and network connections.

* This material is based upon work supported by the Chancellor's Research Fund and UMass Joseph P. Healey Endowment Grants, and the Research Seed Initiative Fund (RSIF), College of Engineering, UMass Dartmouth.

Conventional firewalls are not sufficient for protecting service-oriented systems because web services attackers can initiate attacks as request/response traffics using HTTP protocol that can pass conventional firewalls. The most commonly used conventional firewalls are package filtering firewalls, stateful inspection firewalls, and application level firewalls [1]. A packet filtering firewall only restricts IP addresses or TCP ports recorded in an IP table; however, the port 80 reserved for HTTP and SOAP traffics cannot be blocked on a server that hosts the web services. Thus, a malicious web service invocation can easily pass a packet filtering firewall. On the other hand, a stateful inspection firewall can keep track of TCP/IP connection states and take actions accordingly, but it does not look into packet contents. Similarly, an application level firewall also blocks only those suspicious network traffics with protocols that might be used by an attacker. For example, an application gateway for an FTP server can be configured to accept FTP traffics only and reject all packets using other protocols. Therefore, both stateful inspection firewalls and application level firewalls are not capable of detecting XML-based attacks, e.g., SQL injection attack and overloaded payload attack, which are embedded in XML-based messages [2, 3].

Lack of effective security mechanisms for web services is one of the major reasons why there are so many organizations hesitating to adopt service-oriented technologies despite their significant advantages. In this paper, we introduce an approach to securing service-oriented systems by developing a state-based XML firewall at the application level. Our approach supports Role-Based Access Control (RBAC) [4] for users and detection of XML-based attacks. The XML firewall design introduced in this paper is based on a formal XML firewall model presented in previous work [5], where access permissions to web services are only granted to users who are authenticated and authorized. We develop a detailed design by defining state-based information, user information, and various access control policies and detection rules. Finally, to demonstatrate the effectiveness of our approach, we implement a prototype state-based XML firewall for efficient detection of XML-based attacks to a hospital management service-oriented system.

2. Related Work

Web services security has been an active research area in recent years. Many organizations such as IBM and Cisco, tried to identify major threats to web services in order to protect service-oriented systems more effectively [6, 7]. Typical XML-based attacks include request flooding attack, SQL injection, parameter tampering, overloaded payload attack, and recursive payload attack [2]. A request flooding attack is a type of XML Denial of Service (XDoS) attack, where the attacker floods the web service provider with a large number of web service requests in order to exhaust the resources at the server side. XDoS attacks are similar to packet-based DoS attacks that flood servers with lots of data (e.g., SYN packets); however, conventional firewalls cannot detect XDoS attacks because XDoS attacks are threats to the availability of web services rather than network connections. An SQL injection attack involves tampering the input fields of database requests in order to obtain unauthorized access to data or stored procedures; while a parameter tampering attack is a process of tampering the method parameters passed to a web service operation, and resulting in undesired service behaviors. An overloaded payload attack and a recursive payload attack can exhaust the XML parser of a service provider by sending huge XML data and embedding deeply nested elements in a web service request, respectively.

There is very little previous work on protecting web service providers from being attacked. Fernandez proposed a pattern-based language for XML firewall [8]. Two patterns for design of XML firewall were proposed, which are security assertion coordination pattern using role-based access control for access to distributed resources, and filter pattern for filtering XML messages or documents according to institution policies. Hoktamp discussed the need for XML firewall and possible techniques to protect web services [9]. He analyzed the security issues at three levels of enterprise application integration, namely intranet, extranet and Internet. Cremonini et al. discussed about integrating XML firewall with existing web services security specifications [10]. They analyzed serious security risks in stateful SOAP protocols such as WS-Reliable Messaging, and presented some design guidelines to develop semantics-aware firewalls that can be integrated with the Web Service Architecture (WSA). Bebawy et al. discussed how to apply business specific rules in a centralized manner to develop a web services firewall, called Netdgy [11]. In their implementation, SOAP messages are removed from the transport layer and examined for attack detection, and then induced back into the OSI stack if the XML message is not corrupt. The Netdgy system only supports prevention of limited types of web services attacks such as buffer overflow and SOAP-based DoS attacks. Furthermore, it does not provide any access control mechanisms for users; instead, it supports authorization based on IP tables, which is in the same manner as a conventional packet filtering firewall where messages originating from certain IP address are either dropped or accepted according to a list of blocked IP addresses. Different from the Netdgy system, our effort is to develop a modularized XML firewall that is customizable and targeted for various types of XML-based attacks, thus our approach provides a more comprehensive solution to web services security.

3. Development of State-Based XML Firewall

3.1 State-Based XML Firewall

Based on the formal model of XML firewall we introduced previously [5], we design the state-based XML firewall as a software module with four functional components, namely client interface, RBAC processor, SOAP filter, and admin interface, which coordinate to protect the web services deployed on a web server. As shown in Figure 1, the four major components in an XML firewall are supported by two databases: *User_Info* database and *State_Info* database, which store user information and state-based information, respectively. The client interface module interacts with web service clients and is responsible for receiving requests and sending responses back to the service clients. The actual web services can be deployed either on the same or a different machine where the XML firewall is installed; however, they can only accept requests from a service client through a service proxy defined in the client interface module. As illustrated in Figure 1, each deployed web service (e.g., *WS*1) has a corresponding web service proxy (e.g., *WS*1*P*) defined in the client interface module. The client interface module provides exactly the same interface for web service invocation as the deployed web services, so it is transparent to the web service clients. A client can access an actual web service only after it successfully passes through the XML firewall because the service proxies are the only interface for web service invocations. Authentication and authorization are the major features of the XML firewall for providing user access control. These features ensure that only valid users are allowed to access services. The login block defined in the client interface module provides a basic mechanism for user authentication; while the RBAC processor is responsible for authorizing a user with predefined roles and access permissions. The RBAC processor can determine whether a client has appropriate permissions to access a web service. If a malicious user is detected for a lack of access permissions, any attempts to access the web service by that user will be denied, and the user will be forced to log out of the system. In order to provide a valid duration for a user to invoke web services, we first define the concept of *user session* as follows.

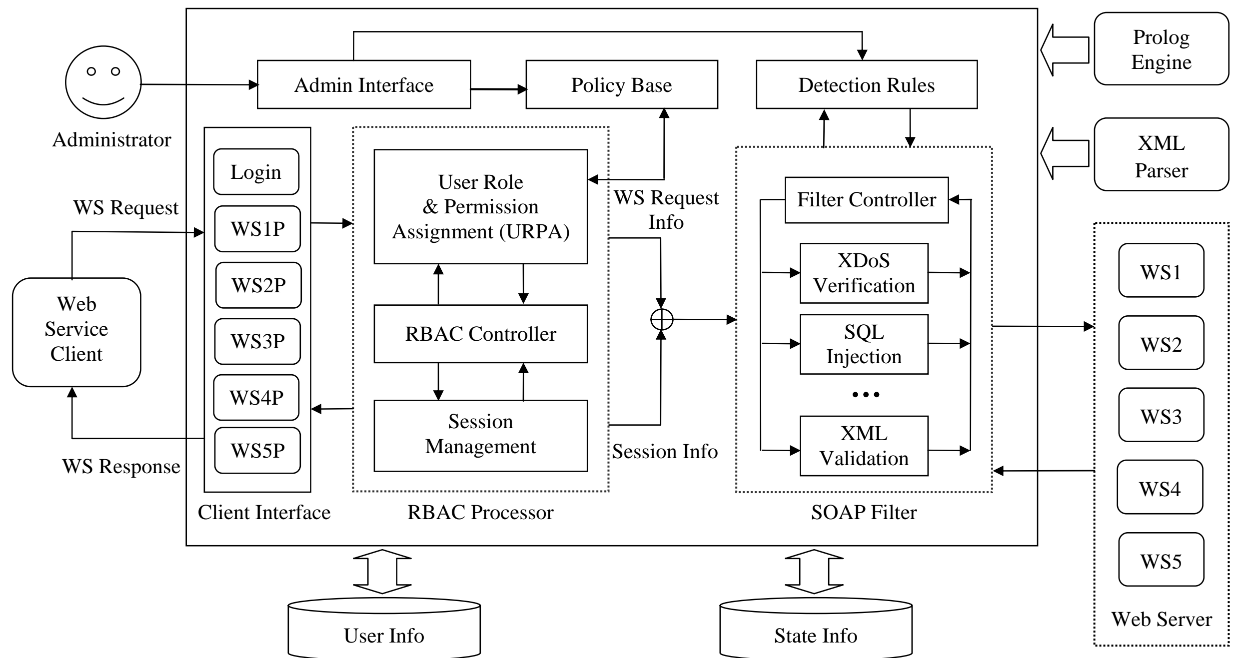

Figure 1. Architectural design of state-based XML firewall

Definition 3.1 A *user session* is defined as a 5-tuple *(UID, SID, RO, ST, ET)*, where *UID* is a user ID, *SID* is a session ID, *RO* is a set of roles that will be assigned to the user, *ST* is the session start time, and *ET* is the session expiration time. A user session is created when the user logs in and destroyed when the user logs out.

After a user logs in and passes the authentication step, his user information is transferred to the RBAC processor module for authorization. Before the User Role and Permission Assignment (URPA) module assigns the user appropriate roles and corresponding access permissions, the session management module in the RBAC processor creates a user session for that user, which has a start time and an expiration time. During the period of time when the session is valid, a user can make requests to web services without being authenticated again. The URPA module, which is used to assign roles to users and permissions to roles, interacts with the *Policy Base*, which is a repository of access control policies defined in Prolog by an administrator through the admin interface. The reasoning process for authorization is supported by a Prolog engine as well as user's information, such as a user's trust level, stored in *User_Info* database.

For every incoming web service request from a user, the RBAC controller verifies whether the associated user session is valid and the user has sufficient permissions to invoke the web service. If the user has enough permission to access the web service, his request in a form of XML message, along with the session information will be passed to the SOAP filter for threat detection and content analysis. Otherwise, the user's request will be denied by the RBAC controller immediately.

The filter controller in the SOAP filter module is responsible for detection of suspicious requests. It examines the session information passed to it as well as the data from the *User_Info* and *State_Info* databases to determine whether the user request is suspicious of any kind of attacks. The detection process is supported by the detection rules defined in Prolog by an administrator, which are modularized into different rule sets for detection of different types of XML-based attacks, e.g., an XDoS attack and an SQL injection attack. Thus, the modularized rule sets can be invoked individually, which support efficient reasoning in real-time. In addition, there is also a set of rules used by the filter controller for detection of attacker suspects. For example, when the filter controller detects a suspicious user with high frequency of requests (determined by predefined thresholds as shown in Section 3.4), the user's request will be passed to the XDoS verification module to verify if the user is performing an XDoS attack. Similarly, if the controller detects that a user request exceeds the normal packet size, the XML message will be sent to the XML validation module to verify for oversized payload attack. On the other hand, if a user request is a normal one, the request will be immediately passed to the web server for web service invocation.

The XDoS verification module requires investigation of a user's previous behavior in order to verify if a user is performing an XDoS attack. If the user has a very low trust level or has been suspected as an XDoS attacker for

a number of times, not only the request from that user will be dropped, but also the user's trust level may be degraded further. Different from the XDoS verification module, the SQL injection module only evaluates the parameters passed to a web service operation by matching them with predefined regular expressions in order to check for any malformed parameters or parameter tampering. Similarly, the XML validation module interacts with the XML parser to evaluate if the request message is well formed by comparing it with an XML schema, and also checks for the size and nesting depth of the message. If any malicious activity is detected and confirmed, the request for the web service invocation is denied immediately; otherwise, the request is passed to the web server for processing. When the service invocation is completed, the result will be forwarded back to the client through the client interface.

One of the major advantages of our approach is that the access policies and the detection rules are modularized; therefore, they can be dynamically updated without recompiling and reinstalling the XML firewall. As shown in Figure 1, a human administrator can add, remove or update any of the access policies and detection rules through the admin interface at runtime. However, during the updating process, the Prolog engine must wait until the updating process is completed.

3.2 Database Design for State-Based XML Firewall

In the detection process, the critical information used by the XML firewall for decision making is the data stored in *State_Info* and *User_Info* databases, which are used for detecting and verifying different types of XML-based attacks. In the following, we give some key definitions of data types used in *State_Info* and *User_Info* databases for detection of XDoS attacks.

Definition 3.2 A *user state* is a 5-tuple *(UID, SID, TR, FR, TL)*, where *UID* is the ID assigned to the user at the time of registration, *SID* is the ID of the session that is initiated, *TR* is the total number of requests made by the user in the current session, *RF* is the request frequency, i.e., the number of requests made by the user in a recent time interval, and *TL* is the user's current trust level.

Definition 3.3 A *firewall state* is a triple *(RE, DE, RT)*, where *RE* is the number of requests that are received by the XML firewall but not yet forwarded to the web server. *DE* is the number of requests that are being processed by the detection modules in the SOAP filter. *RT* is the number of requests in a recent time interval, e.g., the last five minutes. A firewall state is a measure of the work load on the XML firewall system.

Definition 3.4 A *web service state* is a triple *(WID, NR, SI)*, where *WID* is the ID of the web service, *NR* is the number of requests currently being processed by the web service, and *SI* is a state indication of the web service, which can be *busy*, *normal* or *free*. The state indication of a web service indicates the work load of the web service that is determined by thresholds set by an administrator.

Definition 3.5 A *user credential* is a 4-tuple *(UN, PW, UID, TL)*, where *UN* is the user name, *PW* is the password specified by the user at registration time, *UID* is the user ID, and *TL* is the current trust level assigned to the user. A user receives a "normal" trust level at the time of registration, and his trust level can be updated later at runtime based on the user's most recent behavior.

Based on the above state-based information and user information, the SOAP filter can detect and verify XDoS attacks in real-time. Note that the databases store not only the current state and user information, but also the previous states and the recent user information that are useful for attack verification.

3.3 Role-Based Access Control Policies

A role is an abstraction that represents a set of permissions that are needed to perform the tasks associated with a position. Role-based authorization policies specify the roles that each user may adopt, and the permissions associated with each role [4, 12]. From earlier research, it has been argued that it is desirable to separate policy from the application code, so policies can be easily changed over time [13]. Therefore, in this project, we choose Prolog as a specification language for both access control policies and detection rules. Prolog is a declarative language, and can be used to specify both facts and production rules or policies. With a solid mathematical foundation, Prolog allows to reason from a set of rules and supports meta-level reasoning, making policy conflict detection possible. Consider the following access control policies. In a hospital management system, a staff member (e.g., a billing clerk) and a pharmacist can only access a patient's contact and billing information but not his medical records. A patient can be assigned to a doctor or a nurse, who may have full access to the patient's medical records and contact information, but not his billing and account information. A patient can access all records of his own, including his contact information, billing and accounts, and medical records. The access control policies can be specified in Prolog as follows.

```
isValidRole(patient).
isValidRole(doctor). isValidRole(nurse).
isValidRole(staff). isValidRole(pharmacist).
assignRole(U,R) :- isValidRole(R).
canInvoke(R,T,billingService,accessBill):-
 contains(R,[staff,pharmacist,patient]),
 contains(T,[normal,high]).
canInvoke(R,T,billingService,computeBill):-
 contains(R,[staff,pharmacist]),
 contains(T,[normal,high]).
canInvoke(R,T,accessService,readRecord):-
 contains(R,[doctor,nurse,patient]),
```

```
 contains(T,[normal,high]).
canInvoke(R,T,accessService,writeRecord,P,U):-
 contains(R,[doctor,nurse]),
 contains(T,[normal,high]), assignPatient(P,U),
 assignRole(P,patient), assignRole(U,R).
canInvoke(R,T,contactService,accessContact):-
 contains(R,[staff,doctor,nurse,patient]),
 contains(T,[normal,high]).
```

Note that in the above Prolog code, *R* and *T* represent a user's role and the trust level of a user, respectively. Any user must take certain role and have at least a "normal" or "high" trust level before he can access any resource. The predicate *isValidRole* lists various roles defined in the system. The predicate *assignRole(U, R)* is true when a user with UID *U* is assigned a valid role *R*. Similarly, *assignPatient(P, U)* is true when the patient with UID *P* is assigned to a doctor or a nurse with UID *U*. The predicate *canInvoke* determines whether a user with a certain role has the permission to invoke a web service operation. For example, the predicate *canInvoke(R, T, accessService, readRecord)* specifies that a user with role *R* and trust level *T* can invoke the web service operation *readRecord* defined in web service *accessService*. Similarly, the predicate *canInvoke(R, T, accessService, writeRecord, P, U)* ensures that a doctor or a nurse *U* can update a patient *P*'s record only if the patient *P* has been assigned to the doctor or nurse with UID *U*.

3.4 Real-Time Detection of XML-Based Attacks

The SOAP filter is responsible for real-time detection of XML-based attacks. The process of detecting XML-based attacks involves two major steps, which are detection of suspicious SOAP messages and verification of attacks. Suspicious SOAP messages are detected by the filter controller, which uses the session and state information to find possible request flooding attacks, uses certain predefined patterns to find matched strings in the parameters passed to a web service operation; and also keeps track of the maximal allowed message size and the maximal allowed nesting depth in the incoming XML messages in order to detect oversized or recursive payload attacks. We now use an XDoS attack as an example to show how to detect XML-based attacks using our state-based XML firewall. To detect XDoS attacks, the filter controller looks into the session information to check if the current frequency of requests (e.g., the number of request during the last minute) made by a certain user exceeds the threshold set by an administrator. If the frequency exceeds the limit, any new requests from that user will be sent to the XDoS verification module for further analysis. Some sample rules used by the filter controller for XDoS detection are illustrated as follows.

```
checkThreshold(W,S,X):- threshold(W,SI,Y),X > Y.
threshold(accessService,busy,20).
threshold(accessService,normal,40).
threshold(accessService,free,60).
```

In the above rules, *W* is the web service name, *S* represents the session ID, and *X* is the number of requests per minute made by a user who is currently under investigation. The predicate *checkThreadhold* evaluates to true when the number of requests made by the user during the last minute exceeds the limit determined by the web service state indication. For this example, the state indication of a web service is *busy*, *normal*, or *free* if the number of requests processed by the web service during the last minute is larger than 40, between 20 and 40, or less than 20, respectively. According to the above rules, when the web service is *busy*, *normal* or *free*, the corresponding limit on number of requests per minute is 20, 40 or 60, respectively. Note that the information about the web service state and the number of requests the user made during the last minute are stored in *State_Info* database. To simplify matters, the threshold in our current XML firewall implementation does not depend on the firewall state that is specified in Definition 3.3.

If a query to the predicate *checkThreshold* returns true, the corresponding request will be passed to the XDoS verification module where the user's violation history is analyzed. The following Prolog rules demonstrate how to verify an attacker and when to degrade a user's trust level.

```
xdosVerify(U,T):- inspectHistory(U,T,V).
inspectHistory(U,T,V):-
  T = high, dataConnect(U,3,V), V = '3',
  degradeTrustLevel(U,normal).
inspectHistory(U,T,V):-
  T = normal, dataConnect(U,5,V), V = '3',
  degradeTrustLevel(U,low).
inspectHistory(U,T,V):-
  T = low, dataConnect(U,7,V), V = '3'.
  degradeTrustLevel(U,permanentlyBlocked)
dataConnect(U,X,V):-
  java_object('DataConnect',[],data),
  data<-getHistorySessionStatus(U,X) returns V.
degradeTrustLevel(U,T):-
  java_object('DataConnect',[],data),
  data <- recordTrustLevel(U,X).
```

The Prolog code inspects a user's violation history of exceeding service invocation frequency threshold. If the user's trust level is "high", the XDoS verification module only checks the user's previous 3 sessions. If the user has 3 violations, his trust level will be degraded to "normal". On the other hand, if the user's trust level is "normal" or "low", then the user's previous 5 or 7 sessions need to be checked. Similarly, when the user reaches the limit of 3 violations, his trust level will be degraded to "low" or "permanentlyBlocked", respectively. In all above cases, if a query to the predicate *xdosVerify* evaluates to true, the user's current session will be immediately closed. In this case, the user must log in again before he can make further requests. Note that the Prolog code listed above requires invoking Java methods *getHistorySessionStatus* and *recordTrustLevel* to acquire information from *State_Info* database, and record a user's trust level as history information in *User_Info* database, respectively.

Another example to show how to detect attacks using our XML firewall is to detect SQL injection attacks. SQL injection is a technique used to exploit the vulnerabilities in web applications that communicate with databases. The basic idea behind SQL injection is to convince the application to run some malicious SQL code that may result in unauthorized data access or data loss. SQL injection attacks mostly occur due to a lack of user input validation. Although SQL injection is a general technique to attack web-based applications, in the context of service-oriented systems, it can tamper web service parameters which are embedded in XML messages. Thus, in this paper, we treat it as a type of XML-based attack. A simple example of SQL injection attack is called concatenated query attack, where the user manipulates a parameter to form a concatenated query. When a normal query "`SELETE * FROM users WHERE userid = 'user1'`" is manipulated to "`SELECT * FROM users WHERE userid = 'user1'; DELETE FROM users; -- x'`", the execution of the query results in data loss.

In our current implementation of XML firewall, the SOAP filter uses regular expressions to specify string patterns such as concatenation of "`';`" and "`';--`". If any input string matches one of the predefined patterns, the user will be detected as an attacker for SQL injection, and the user's current session will be closed immediately.

4. A Case Study

In this section, we use a case study to demonstrate how a state-based XML firewall can be used to effectively detect XML-based attacks. We developed a prototype XML firewall, and installed it on the same machine where a service-oriented system was deployed. The service-oriented system we adopted in this case study is a hospital management system, where different roles and access control policies are defined to determine a user's access permission to specific services. The hospital management system is an adaptation of the system presented in previous work [13], which is implemented as a service-oriented system. The related user roles as well as their corresponding access permissions are the same as those defined in Section 3.3. We now first simulate an SQL injection attack by accessing the web service *accessService*, which allows a user with sufficient permissions to write medical records for a patient. Consider *User1* with a patient role who is assigned to nurse *User2*. Since *User1* is assigned to *User2*, *User2* has permission to write *User1*'s medical records by invoking *writeRecord* operation defined in web service *accessService*. The invocation requires four parameters, namely the ID of the user who writes the record, the ID of the patient whose record is to be updated, a string containing medical report information, and the type of the report. A legitimate request from the nurse could be *writeRecord("User2", "User1", "The patient reacted abnormally to new drugs.", "Observation")*, which results in an SQL query as follows.

```
INSERT INTO patientRecords VALUES('User2',
'User1', 'The patient reacted abnormally to new
drugs.','Observation');
```

Now a malicious user may perform an SQL injection attack by tampering the parameters in the web service invocation. *User2* may send the fourth parameter as "`Observation'); DELETE FROM users; --dummystring`". The resultant query in the web service will delete all the records in *users* table if the server allows execution of multiple queries.

With the installed XML firewall, when *User2* makes such a request, the XML firewall can successfully detect the SQL injection attack and prevents unauthorized data access by checking the parameters of the request against predefined regular expressions. Figure 2 is a snapshot of the log information showing the successful detection of a simulated SQL injection attack.

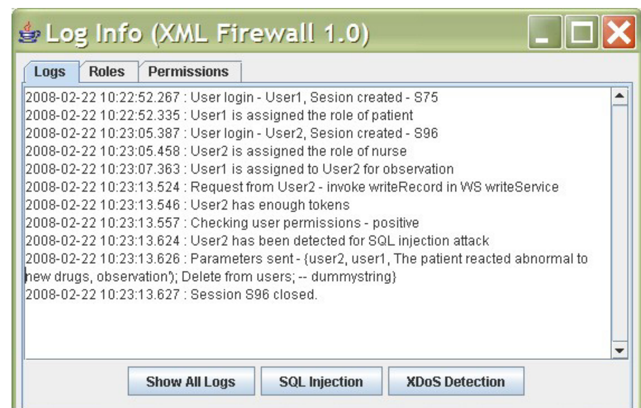

Figure 2. Log information for SQL injection detection

To demonstrate that our prototype XML firewall can effectively detect and prevent XDoS attacks, we simulate request flooding attacks on a web service with a large number of requests from an attacker, and record the response behavior of the server for requests from a normal user. We choose the report generation service implemented in the service-oriented hospital management system because it consumes significant amount of memory space and CPU time. The web service takes around 10 seconds to process a request; thus, the normal response time should be around 10 seconds. We now set up the flooding attack with a number of threads, each of which sends web service requests continuously to the report generation service. When the XML firewall was disabled, we observed that when the number of requests received by the server increases, the response time of a request from a normal user increases significantly. When the frequency of requests reaches around 128 per minute, the web service becomes unavailable to the normal user because the server crashes due to a heap space error. This

is illustrated by one of the curve (denoted as "without XML Firewall") in Figure 3. When we enable our XML firewall and set an appropriate threshold, the web server can be successfully prevented from crashing. Figure 3 shows two other curves that represent the experimental results with the XML firewall enabled when the thresholds for the firewall with *free* state indication are set to 80 and 60, respectively. As shown in Figure 3, when the threshold is 80, the worst response time is 25 seconds, but it drops to normal response time when the attacker increases the request frequency further. To enhance performance, we lower the threshold from 80 to 60, and the worst response time now becomes 17 seconds.

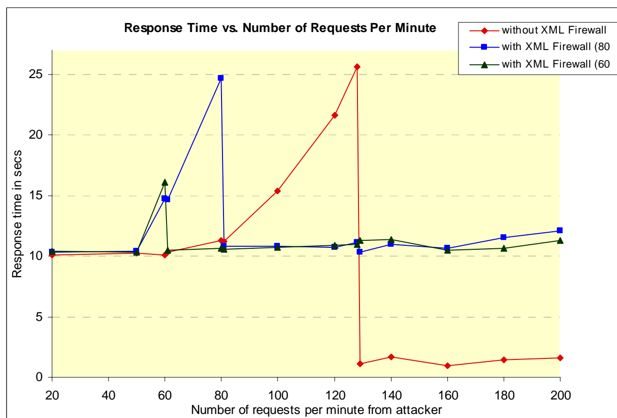

Figure 3. Experimental results for XDoS attacks

Note that a very high threshold could overload the system while a very low threshold might block legitimate users with high request rates. Thus, it is important for the administrator to choose an appropriate threshold for the XML firewall in order to make it work efficiently.

5. Conclusions and Future Work

Service-oriented systems are increasingly deployed over the Internet due to their standardized protocols and techniques that enable the efficient integration of loosely coupled applications over networks. However, due to the open interface for service-oriented architecture, attacks on service-oriented systems are more complicated than traditional attacks that can be handled by conventional firewalls. Thus, there is a pressing need to introduce new security mechanisms to protect service-oriented systems. In this paper, we introduced a state-based XML firewall, which can be used to protect service provider from various XML-based attacks. We developed a detailed design of our state-based XML firewall, and implemented a prototype XML firewall. Our experimental results show that our prototype XML firewall can effectively protect web services from various XML-based attacks. In our future work, we will study new types of XML-based attacks and show how their corresponding attack verification modules can be easily integrated into our current implemented system due to the modular design. We will also consider adopting agent-based technology to provide more intelligence in XML firewall for efficient detection and verification of XML-based attacks.

References

[1] E. B. Fernandez, M. M. Larrondo-Petrie, N. Seliya, N. Delessy-Gassant, and M. Schumacher, "A Pattern Language for Firewalls," In M. Schumacher, *et al.* (Eds.), *Security Patterns: Integrating Security and Systems Engineering*, Wiley, March 2006.

[2] E. Moradian and A. Håkansson, "Possible Attacks on XML Web Services," *International Journal of Computer Science and Network Security (IJCSNS)*, Vol.6, No.1B, January 2006, pp. 154-170.

[3] M. Andrews and J. A. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley Professional, February 2006.

[4] H. Feinstein, R. Sandhu, E. Coyne, and C. Youman, "Role-Based Access Control Models," *IEEE Computer*, Vol. 29, No. 2, 1996, pp. 38-47.

[5] H. Xu, M. Ayachit and A. Reddyreddy, "Formal Modeling and Analysis of XML Firewall for Service Oriented Systems," *International Journal of Security and Networks (IJSN)*, Vol. 3, No. 3, 2008.

[6] P. Crocker and B. Thompson, "Integrating WebSphere DataPower SOA Appliances with WebSphere MQ," *Technical Report*, IBM Hursley Software Lab, March 2007.

[7] Reactivity, "Architecting the Infrastructure for SOA and XML," *White Paper*, Cisco Systems, Inc. 2007.

[8] E. B. Fernandez, "Two Patterns for Web Services Security," In *Proceedings of the 2004 International Symposium on Web Services and Applications (ISWS'04)*, Las Vegas, Nevada, 2004.

[9] M. Holtkamp, "The Role of XML Firewalls for Web Services," *The 1st Twente Student Conference on IT*, Track B, June 2004.

[10] M. Cremonini, S. Vimercati, E. Damiani, and P. Samarati, "An XML-Based Approach to Combine Firewalls and Web Services Security Specifications", In *Proceedings of the 2003 ACM Workshop on XML Security*, Fairfax, Virginia, 2003, pp. 69-78.

[11] R. Bebawy, H. Sabry, S. El-Kassas, Y. Hanna, and Y. Youssef, "Nedgty: Web Services Firewall," In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, 2005, pp. 597-601.

[12] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 4, No. 3, August 2001, pp. 224-274.

[13] M. Y. Becker and P. Sewell, "Cassandra: Flexible Trust Management, Applied to Electronic Health Records," In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, 2004, pp. 139-154.