Defending Against XML-Based Attacks Using State-Based XML Firewall

Haiping Xu, Abhinay Reddyreddy, and Daniel F. Fitch
Computer and Information Science Department
University of Massachusetts Dartmouth, North Dartmouth, MA 02747, USA
Email: {hxu, g_areddyreddy, daniel.fitch}@umassd.edu

***Abstract*—With the proliferation of service-oriented systems and cloud computing, web services security has gained much attention in recent years. Web service attacks, called XML-based attacks, typically occur at the SOAP message level, thus they are not readily handled by existing security mechanisms such as a conventional firewall. In order to provide effective security mechanisms for service-oriented systems, XML firewalls have recently been introduced as one of the major means for web services security. In this paper, we present a framework for state-based XML firewall, called S-Wall, which supports dynamic role-based access control (D-RBAC) and detection of XML-based attacks in real-time. We provide a detailed design of the S-Wall security model by defining state-based information, user information, access control policies, and detection and verification (D&V) rules. The D&V rules are modularized into separate units, which support real-time detection and verification of various types of attacks using state-based information. To illustrate the effectiveness of our approach, we develop a prototype S-Wall, and utilize a case study to demonstrate how S-Wall can be used to efficiently detect and defend against XML-based attacks.**

***Index Terms*—State-based XML firewall (S-Wall), web services security, service-oriented architecture, dynamic role-based access control (D-RBAC), XML-based attack, detection and verification (D&V)**

I. INTRODUCTION

Service-oriented architecture (SOA), as a promising system development paradigm, is defined as an inherently interoperable architecture, which enables interoperability across different enterprise and business solutions [1]. SOA allows the construction of systems using reusable components with well-defined service interfaces, where components can be published as discoverable services over the Internet based on their capabilities. Recently, cloud computing has been proposed as a broad movement to enable interactions among service providers and service consumers using the Internet [2, 3]. Cloud computing paradigm supports not only storage service and platform as a service (PaaS); it also supports software as a service (SaaS) [4]. In this sense, cloud computing follows the philosophy of service-oriented computing, and defines a more comprehensive framework for service-oriented systems. While the industry and government are quickly moving towards SOA and cloud computing paradigm, trust and security issues in service-oriented systems become one of the primary concerns [5].

The key to protect a service-oriented system from XML-based attacks is to understand its threat profile, and to study how threats may affect the performance of the service-oriented system. Many organizations such as IBM and Cisco, attempted to identify major threats to web services in order to protect service-oriented systems more effectively [6, 7]. Threats to web services are typically XML-based attacks that can perform on web services through SOAP (Simple Object Access Protocol) messages, which rely on eXtensible Markup Language (XML) as its message format and application layer protocols such as HTTP for message negotiation and transmission. Typical XML-based attacks include XPath injection attack, XML-based denial of service (XDoS) attack, overloaded payload attack, recursive payload attack, parameter tampering attack, XML injection attack, SQL injection attack, and schema poisoning attack [8-11]. For example, an XPath injection attack takes advantage of the weakness of an XPath parser of a service provider to allow malicious XPath queries on URLs, forms, or other methods in order to gain access to privileged information or unauthorized information; an XDoS attack is a type of request flooding attacks, where an attacker directs malicious traffic to a web service to exhaust the resources at the server side; and an overloaded payload attack can exhaust the XML parser by sending huge XML data embedded in SOAP messages as web service requests. An XML-based attack can also be in a form of a distributed multi-phased attack, e.g., an XML-based Mitnick attack is adapted from the Mitinick attack, which can be used in conjunction with the XML injection attack against web services [12]. Most of the XML-based attacks are not well understood, and their performance has not yet been carefully studied. In order to effectively protect service-oriented systems from XML-based attacks, it is vital to thoroughly investigate the characteristics and behaviors of such attacks and justify their undesired effects on service-oriented systems. Once we are able to successfully simulate major threats to service-oriented systems, the simulated XML-

Manuscript received January 11, 2011; revised February 14, 2011; accepted February 14, 2011.
Corresponding author: Haiping Xu, Email: hxu@umassd.edu

based attacks can be used to evaluate the effectiveness of security mechanisms for service-oriented systems. Thus, effective simulations of major XML-based attacks such as XPath attack, XDoS attack, and overloaded payload attack, are necessary efforts for justifying the usefulness of our proposed security mechanism.

Since XML-based attacks can be initiated as request/response traffic using HTTP protocol that is typically not blocked by a conventional firewall, conventional firewalls are not sufficient for protecting service-oriented systems from XML-based attacks. The most commonly used conventional firewalls are package filtering firewalls, stateful inspection firewalls, and application level firewalls [13]. 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 traffic is typically not blocked on a server where web services are deployed. Thus, a 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 only blocks 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 only to accept FTP traffic 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., an overloaded payload attack), which are embedded in XML-based messages [14, 15].

Many security standards have been developed for protecting web services, but they are still vulnerable to a variety of attacks such as an XDoS attack. Lack of effective security mechanisms for web services is one of the major reasons why some organizations hesitate to adopt service-oriented technologies despite their many advantages. In this paper, we introduce an approach to defending against XML-based attacks at the application level using a state-based XML firewall, called *S-Wall*. Our approach supports dynamic role-based access control (D-RBAC) for users and detection of XML-based attacks in real-time. The design of the S-Wall security model introduced in this paper is based on a formal XML firewall model presented in previous work [16], where access permissions to web services are only granted to those users who are authenticated and authorized. This work also extends our previous efforts on prototyping state-based XML firewall [15] by providing dynamic RBAC mechanism and a reasoning engine for real-time detection of XML-based attacks. Furthermore, we utilize a comprehensive case study to demonstrate how to simulate, detect and defend against two major XML-based attacks, namely, XPath injection attack and a hybrid XDoS and overloaded payload attack. Our experimental results show that the S-Wall security model provides an effective way to protect service-oriented systems from XML-based attacks.

The rest of this paper is organized as follows. In Section II, we describe related work and highlight the relationships to our research. In Section III, we first provide a motivating example of complex XML-based attacks, and then we present a framework for S-Wall security model. In Section IV, we give a detailed design of S-Wall by defining state-based information, user information, access control policies, and detection and verification (D&V) rules. The D&V rules are defined and modularized into separate units, which support real-time detection and verification of various types of attacks using state-based information. In Section V, we utilize a case study to show how major XML-based attacks can be simulated, and how our approach can be used to effectively defend against them. In Section VI, we provide a brief conclusion and mention future work.

II. Related Work

Web services security has been an active research area in recent years [11, 17]. However, there is still very little previous work on protecting web service providers from being attacked. Fernandez *et al.* proposed a pattern-based language for XML firewall [18, 13]. 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 [19]. He analyzed the security issues at three levels of enterprise application integration, namely intranet, extranet and Internet. Cremonini *et al.* attempted to integrate XML firewall with existing web services security specifications [20]. 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 service firewall, called Netdgy [21]. 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 IP table based authorization, which is in the same manner as a conventional packet filtering firewall where messages originating from a certain IP address are either dropped or accepted according to a list of blocked IP addresses. Different from the Netdgy system, our approach is to develop a modularized state-based XML firewall that is customizable for defending against various XML-based attacks. Thus, our approach provides a more comprehensive solution to web services security.

In addition, there are currently a few products available on the market for XML-based protection of web services. The Forum XWALL [22] and IBM DatapowerXS40 XML Gateway [6] are the typical examples that provide interfaces for users to define policies. In these products,

policies can be defined through a GUI or using XSLT (EXtensible Stylesheet Language Transformations) as in XWALL or DatapowerXS40, respectively, and can be used for content inspection against malicious SOAP messages. Although these products provide some level of security to web services, their functionalities are still very limited. For example, the above approaches are not state-based, so they cannot effectively protect web services from state-based attacks (e.g., an overloaded payload attack on a web server with a heavy load). Furthermore, they typically do not support role-based user authorization, and thus, unauthorized user may access web services with insufficient permissions. In contrast, our approach provides a more general solution to implementing state-based XML firewalls, which supports both role-based user access control and real-time detection of XML-based attacks.

On the other hand, an intrusion detection system (IDS) is complementary to the firewall approach by monitoring and detecting intrusions to systems and networks in real-time [23-26]. Although there are previous efforts to build IDS using SOA [27-29], to the best of our knowledge no attempts on using IDS to secure service-oriented systems from XML-based attacks have been reported. This is not only because SOA is a relatively new architecture that is becoming more and more popular in recent years, but also because XML-based attacks typically occur at the SOAP message level, they are not readily to be detected by monitoring audit data or network traffic using existing IDS approaches. In contrast, our proposed S-Wall security model aims at protecting service providers from XML-based attacks – it not only behaves as an application layer firewall, but also utilizes existing IDS analysis approaches such as the misuse detection method for detection of XML-based attacks in real-time.

In our previous work, we introduced a Petri net based formal model of XML firewall and verified its correctness using an existing Petri net tool [16]. The formal model supports user authentication and role-based user authorization according to policy rules that can be updated dynamically. In this paper, we use the formal model as a high-level design, and present a framework for S-Wall. We develop the detailed design of the S-Wall security model and use a case study of a financial management service-oriented system to demonstrate the effectiveness of our S-Wall approach.

III. Defending Against XML-Based Attacks

A. A Motivating Example

Consider a scenario where a financial company has established a set of stock market services (SMS) to allow users to query the stock prices and trade stocks during the trading day. The users require real-time up-to-date information to make educated decisions on their financial investments. Certain traders would also need this information quickly and frequently, so that they can respond to emerging changes in a stock. As shown in Fig. 1, a user can access the stock information service (SIS) and stock trading service (STS) through a real-time stock information and trading application (RT-SIT). A stock advisor may provide stock trading advising service to a user through the same RT-SIT application. In addition to the stock prices, SMS can provide recommendations on stock market trading, which require dynamic invocation of the stock recommendation service (SRS). The SRS is designed as an internal service because it can only be accessed by a certified stock analyst or a system administrator. Since different certified stock analysts provide their own SRSs that are typically deployed on different machines, the endpoint address of a currently adopted SRS must be configured by an administrator in an XML settings file. In addition, the XML settings file also contains authentication information of users and stock advisors. Thus, at runtime, the SIS needs to make an XPath query to authenticate a user or a stock advisor, read the location of the adopted SRS from the XML settings file, and invoke that SRS dynamically. Fig. 1 also shows the dependency of the provided SMS services. For example, SIS requires invocation of SRS in order to provide stock users stock information including recommendations on stock options, and STS requires invocation of SIS because stock trading is based on real-time stock information available through SIS. In addition,

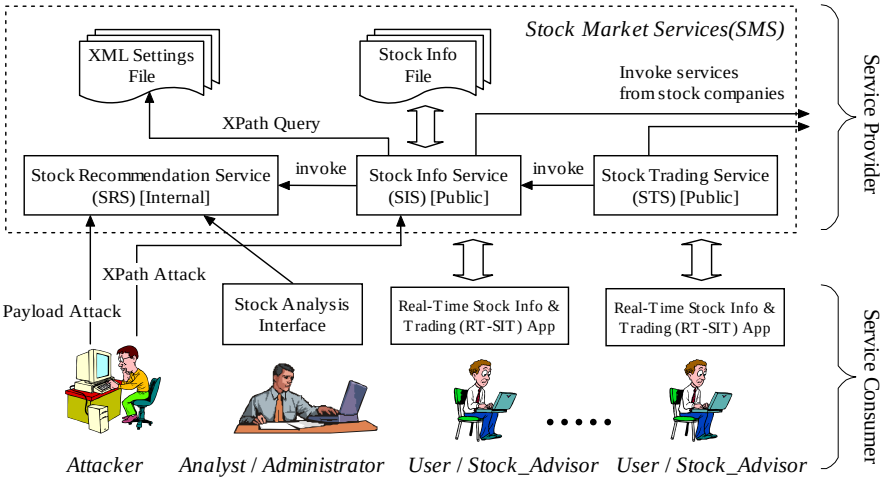

Figure 1. XML-based attacks on stock market services

both SIS and STS require invocation of remote services provided by various stock companies who have contracts with the financial company.

Now suppose there is an attacker who attempts to compromise the SMS. The attacker first tries to attack SIS or STS using an overloaded payload attack or XDoS attack; however, since these services were designed for concurrent access with a large volume of data by many users, they are typically very robust. Upon failure on such attempts, the attacker turns his attention to the internal service SRS. As expected, internal services are not published to the public and can be accessed only by authorized personnel, thus the information for invocation of SRS is unknown to the attacker. In order to capture such information, the attacker pretends to be a normal user and utilizes an XPath injection attack to the public service SIS. The XPath attack brings the attacker pertinent information regarding the internal service SRS such as its endpoint address, credentials to access the service, and connection timeout and limits that should be imposed on the response. Based on the captured information, the attacker performs a hybrid of XDoS/overloaded payload attack to compromise SRS by combining elements of an XDoS attack with elements of an overloaded payload attack. The attacker utilizes a multi-threaded application to generate many requests that contain very large headers in a very short time. These attacks compromise both the memory and CPU of the computer where the currently adopted SRS is deployed. This would result in very slow response or even failure of the SRS, which makes the performance of SIS and STS that depend on SRS significantly affected.

Since the above attack consists of two major steps, namely XPath injection attack and a hybrid of XDoS/overloaded payload attack, we call such attack a two-phased XML-based attack. In order to provide protections to the SMS from such an attack, we introduce a framework for state-based XML firewall (S-Wall) in the following section. Although an S-Wall looks similar to an application firewall, it provides much more comprehensive functionalities for detecting and defending against XML-based attacks in real-time.

B. A Framework for State-Based XML Firewall

The general architecture of the S-Wall model is illustrated in Fig. 2. As shown in the figure, an S-Wall sits between service consumers and a service provider, and can be installed either on the same or a different machine where the actual web services are deployed. An S-Wall interacts with service consumers through its client interface (CI) module, which is responsible for receiving requests from and sending responses back to the service consumers. The CI module is configured such that it can only accept requests from a service consumer through a service proxy defined in the CI module. For example, when web service *WS*1 is deployed, a corresponding service proxy *WS*1*P* is automatically generated in the CI module. The CI module is transparent to a service consumer since it provides exactly the same interface of the deployed web services; a service consumer, however, can access an actual web service only after it successfully passes through the S-Wall.

As shown in Fig. 2, there are two major databases supporting S-Wall, namely *User_Info* database and *State_Info* database, which store user information and state-based information, respectively. In the S-Wall security model, authentication and authorization are the major features for providing user access control, which ensure that only valid users are allowed to access certain web services. For example, the *Login* service defined in the CI module provides a basic mechanism for user authentication, and the dynamic role-based access control (D-RBAC) module is responsible for authorizing a user with predefined user roles and access permissions. The D-RBAC module is also responsible for determining whether a consumer has appropriate permissions to access a web service at runtime. If a malicious user is detected for a lack of access permissions, any attempts to

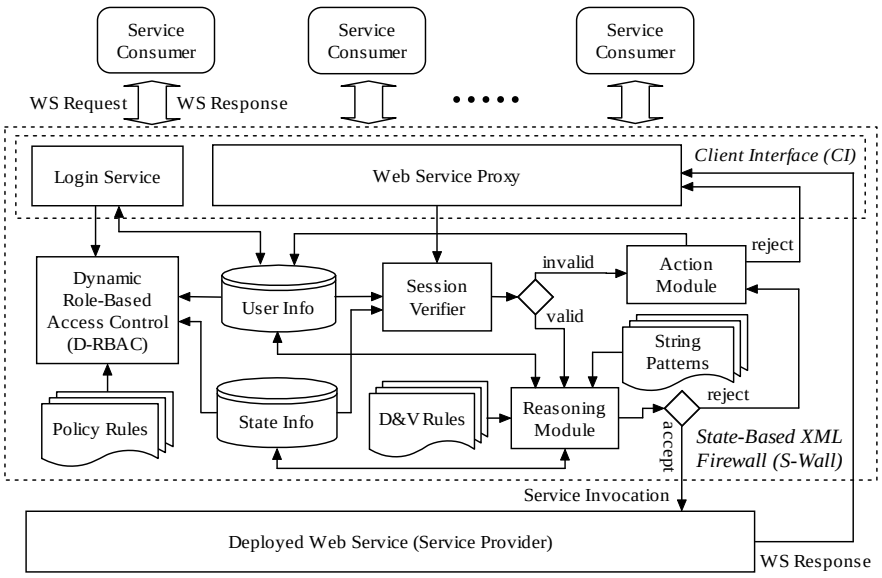

Figure 2. A framework for state-based XML firewall

access web services by that user will be denied, and the user will be forced to log out of the system. Conventional RBAC mechanism has been proposed as one of the most attractive solutions to providing security features in different distributed computing infrastructure [30]. In an RBAC model, users are assigned roles with permissions, which are access modes that can be exercised on a particular object in the system. RBAC mechanism ensures that only authorized users are given access to certain data or resources. Most of the conventional RBAC models follow the same basic structure of subject, role and privilege. However, in a more sophisticated RBAC model, access decisions for an application depend on the combination of a user's credential, the context, the system state, and other factors such as relationship, time and location [31]. The concept of dynamic RBAC mechanism was first introduced in our previous work [32]. It is adopted in our S-Wall security model because S-Wall requires that the access control process should depend not only on a user's identity, but also on the user's current state. In other words, in our S-Wall model, we define certain policy rules that specify a user's access right to web services with certain permissions (restrictions) based on the user's current state and system state. Thus, our S-Wall approach is *stateful*.

In order to provide valid execution duration for a user to invoke a web service, we formally define the concept of user session as follows.

Definition 3.1 User Session. A user session is defined as a 6-tuple (*UID*, *SID*, *RO*, *AP*, *ST*, *ET*), where *UID* is a user ID, *SID* is a session ID, *RO* is a set of roles assigned to the user, *AP* is a set of access permissions that must be updated at runtime, *ST* is the session start time, and *ET* is the session expiration time. A user session is created when a user logs in, and destroyed when the user logs out or when the S-Wall rejects any of the user's requests.

Once a user has logged in and been authenticated, his user information is transferred to the D-RBAC module for authorization. The D-RBAC module then interacts with the *Policy Rules* module, which is a repository of access control policies defined in Prolog, for role assignment and determination of access permissions. The process for authorization is supported by information about a user, such as a user's trust level, stored in the *User_Info* database, as well as state information, such as how often the same request has been made, stored in the *State_Info* database. After the role and access permission assignment is done, a user session is created for that user, which has a start time and an expiration time, and is recorded in the *User_Info* database. During the period of time when the session is valid, a user can make requests to web services without being authenticated again. For every incoming web service request from a user, the *Session Verifier* checks whether the associated user session is valid and the user has sufficient permissions to invoke the web service based on state information. If the user has enough permission to access the web service, his request in a form of XML message, along with the session information is passed to the *Reasoning Module* for threat detection and content analysis. Otherwise, the user's request is sent to the *Action Module*, where a reject message is generated and sent to the corresponding service proxy. In this case, the user session is destroyed by the S-Wall, and this information is recorded into the *User_Info* database.

When the reasoning module starts to detect malicious requests, it first examines the session information passed to it as well as the related data from the *State_Info* databases to determine whether the user request is suspicious of any kind of XML-based attacks. The detection process is supported by the *D&V Rules* module, which consists of detection and verification rules defined in Prolog by system administrators. The detection rules are used to detect suspicious requests only; while the verification rules are modularized into different rule sets for verification of different types of XML-based attacks, e.g., an XDoS attack. Thus, the modularized rule sets can be invoked individually, and support efficient reasoning in real-time. For example, when the reasoning module detects a suspicious user with high frequency of requests (determined by a predefined threshold as shown in Section IV-C), the user's request will be verified using XDoS verification rules for possible XDoS attack. Similarly, if the reasoning module detects that a user request exceeds the normal packet size, the XML message will be verified for overloaded payload attack. On the other hand, if a user request is not a suspect of any kind of attacks, the request will be immediately passed to the corresponding deployed web service for service invocation. Note that verification of XML-based attacks (e.g., an XDoS attack) requires investigation of a user's previous behaviors. 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 detection, detection of injection attacks (e.g., an XPath injection attack) only requires evaluating the parameters passed to a web service operation by matching them with predefined string patterns to identify any malformed parameters or parameter tampering. If any malicious activity is detected and confirmed, the service request is sent to the action module, where a reject message is generated and sent to the corresponding service proxy. In this case, the user session is destroyed, and such information is recorded into the *User_Info* database. On the other hand, if the request is accepted, it is passed to the deployed web service for service invocation, and the result is forwarded back to the service consumer through CI.

IV. Design of State-Based XML Firewall

A. Definitions of User and State Information

In the detection and verification process, the critical information used by the S-Wall for decision making is the data stored in the *State_Info* and *User_Info* databases. The state-based information and user information are used by the S-Wall to detect and verify different types of XML-based attacks. As an example, we now consider XDoS attack and overloaded payload attack, and give

some key related definitions of the data types used in the *State_Info* database as follows.

Definition 4.1 User State. 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 session ID 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 predefined time interval (e.g., one minute), and *TL* is the user's current trust level, which can be *high*, *normal*, *low* or *permanentlyBlocked*.

Definition 4.2 Firewall State. A firewall state is a 4-tuple (*RE*, *DE*, *RT*, *SEC*), where *RE* is the number of requests that are received by the S-Wall but not yet forwarded to the web server, *DE* is the number of requests that are being processed by the reasoning module, *RT* is the number of requests in a recent predefined time interval, and *SEC* is a current security level of the firewall, which can be *green*, *yellow* or *red*.

Definition 4.3 Service State. A service state is a 7-tuple (*SID*, *NR*, *AT*, *RT*, *NP*, *SR*, *SI*), where *SID* is the service ID, *NR* is the number of requests currently being processed by the web service, *AT* is a list of attacks that have been performed on the service, *RT* is the average response time during the recent predefined time interval, *NP* is the number of overloaded payload attacks detected during the recent predefined time internal, *SR* is the average size of SOAP requests during the recent predefined time interval, and *SI* is a state indication of the web service, which can be *busy*, *normal* or *free*.

We also give some key definitions of the data types used in the *User_Info* database for detection and verification of XDoS attack and overloaded payload attack as follows.

Definition 4.4 User Credential. 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 behaviors or activities.

Definition 4.5 Active User. An active user is a triple (*UID*, *RO*, *TO*), where *UID* is the ID of the user who has logged into the S-Wall, *RO* is the set of roles assigned to the user, and *TO* is the number of tokens assigned to the user when the user logs in. Note that the tokens assigned to a user are used to control the maximal number of requests that the user can make in each valid session.

Based on the above state-based information and user information, the reasoning module can detect and verify XDoS attack and overloaded payload attack in real-time. The corresponding tables are created in the *State_Info* and *User_Info* databases, which are used to store not only the current state and user information, but also the previous states and recent user information that are useful for attack detection and verification.

B. 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 [30, 33]. 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 [34]. Furthermore, a policy language must be expressive so that the intended rules can be written naturally, easy to understand and effectively computable. There has been some previous research in defining policy languages [34-36]. Most of the policy languages such as Delegation Logic [35], Binder [36], and Cassandra [34] are based on DataLog, where a DataLog statement can be easily translated into declarative English sentences. This helps users to formulate security policies that capture their intentions accurately. However, an obstacle to deployment of trust management systems with Datalog-based policy languages is that there is a lack of tool support for such languages. In this paper, we choose Prolog as a specification language for both access control policies and D&V rules for the following reasons. 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 based on the motivating example presented in Section III-A. In the stock information & trading system, any user (registered or unregistered user), analyst, stock advisor, and administrator can access real-time stock information, but only an analyst or an administrator can update stock recommendation information. Both a registered user and an administrator can access trading transaction history, but only a registered user is allowed to initiate a transaction for stock trading. The system also provides stock training sections, where a registered user is assigned to a stock advisor who is responsible for giving training lessons on the operations of the system and providing financial advices to the user. The above access control policies can be specified in Prolog as follows.

```
isValidRole(unregisterUser).
isValidRole(registeredUser).
isValidRole(stockAdvisor).
isValidRole(stockAnalyst).
isValidRole(administrator).
assignRole(U,R) :- isValidRole(R).
canInvoke(R,T,stockInfoService,getQuote):-
  contains(R,[stockAnalyst,administrator,
              stockAdvsior,registerUser,
              unRegisteredUser]),
  contains(T,[normal,high]).
canInvoke(R,T,stockInfoService,updateStockRec
omInfo):-
  contains(R,[analyst,administrator]),
  contains(T,[normal,high]).
canInvoke(R,T,stockTradingService,readTransHi
strory):-
  contains(R,[registeredUser,administrator]),
  contains(T,[normal,high]).
canInvoke(R,T,stockTradingService,initiateTra
nsaction):-
  contains(R,[registeredUser]),
  contains(T,[normal,high]).
```

```
canInvoke(R,T,stockTradingService,stockTraini
ng,U,A):-
  contains(R,[stockAdvisor]),
  contains(T,[normal,high]),
  assignAdvisee(U,A),
  assignRole(U,registeredUser),
  assignRole(A,R).
```

In the above Prolog code, `R` represents a user's role, and `T` represents the trust level of a user. Any user must take a certain role and maintain at least a `normal` or `high` trust level before he can access some 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, `assignAdvisee(U,A)` is *true* when a registered user with UID `U` is assigned to a stock advisor with UID `A`. The predicate `canInvoke` determines whether a user with a certain role has the permission to invoke an operation defined in a web service. For example, the predicate `canInvoke(R,T,stockInfoService,accessStockInf o)` specifies that a user with role `R` and trust level `T` can invoke the web service operation *accessStockInfo* defined in web service *stockInfoService*. Similarly, the predicate `canInvoke(R,T,stockTradingService,stockTraini ng,U,A)` ensures that a stock advisor with UID `A` and a trust level `normal` or `high` can train a registered user `U` only if the user `U` has been assigned to stock advisor `A`.

Note that our D-RBAC approach is dynamic, which means the access control permissions assigned to a user can be changed at runtime based on user states. For example, according to the aforementioned access control policies, when a registered user's trust level is `normal` or `high`, the user is allowed to initiate a transaction on STS. However, when the registered user's trust level is downgraded from `normal` to `low`, the user's access to STS for initializing transactions will be denied.

C. Real-Time Detection of XML-Based Attacks

The reasoning module is responsible for real-time detection and verification of XML-based attacks by checking the SOAP message as well as the parameters passed to a web service operation. The reasoning engine algorithm adopted by the reasoning module for decision making is presented in Algorithm 1. As shown in the algorithm, when a SOAP message with a valid user session is sent to the reasoning module, the reasoning engine first verifies if the parameters involves any injection attack (e.g., an XPath injection attack or an SQL injection attack) by checking against predefined string patterns that define possible threats to parameters. If an injection attack is detected, the SOAP message is rejected, and the information related to the attack is recorded into the *User_Info* and *State_Info* database. Note that the detection of an injection attack does not require any user information or state information from the two major databases. On the other hand, if the reasoning module concludes that the parameters are not compromised, it starts to detect other types of XML-based attacks by investigating the SOAP message content. The process of detecting other types of XML-based attacks involves two major steps, which are detection of suspicious SOAP messages and verification of attacks. Suspicious SOAP messages can be detected using the session and state information by the reasoning module. For example, in order to detect XDoS attack and overloaded payload attacks, the reasoning module attempts to find possible flooding requests, and keep track of the maximal allowed message size and the maximal allowed nesting depth in the incoming XML messages. If a certain type of attack is detected, the reasoning module will attempt to verify the attack using additional evidence from the *User_Info* and *State_Info* database. Once an attack is confirmed, the SOAP message is rejected, and sent to the action module, where a rejection message is generated and sent back to the service consumer through the service proxy. In addition, the *User_Info* and *State_Info* database are updated accordingly with the information related to the attack. Otherwise, the SOAP message is sent to the deployed web service for service invocation, and after the service invocation, the results are sent back to the service consumer through the service proxy.

Algorithm 1: Reasoning Engine

Input: SOAP message with parameters for service invocation
Output: *accept* / *reject*

```
1. Verify parameters for injection attacks using string patterns
2. if pattern matched
3.    then update User_Info and State_Info database
4.       output reject to the action module
5. else detect suspicious SOAP message using detection rules
6.    if SOAP message is suspicious of any attack
7.       then switch (attack_type)
8.          case XDoS_attack:
9.             verify XDoS attack using XDoS verification rules
10.         case Overloaded_payload_attack:
11.            verify Overloaded_payload_attack using Payload
12.            verification rules
13.         case ... // other types of XML-based attacks
14.      if any attack confirmed
15.      then update User_Info & State_Info database
16.            output reject to the action module
17.      else output accept and invoke a deployed web service
18.   else output accept and invoke a deployed web service
```

We now use the XDoS attack as an example to show how to detect XML-based attacks using S-Wall. To detect XDoS attacks, the reasoning module 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 predetermined by an administrator. If the frequency exceeds the limit, any new requests from that user must be checked further using XDoS verification rules. Some sample rules used by the reasoning module 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 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 `checkThreshold` evaluates to be *true* when the number of requests made by the user during the last minute exceeds the limit determined by the service state indication `SI`. In this example, the state indication of a web service is set as *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 the *State_Info* database. To simplify matters, in our current S-Wall implementation, the threshold does not depend on the firewall state that is specified in Definition 4.2.

If a query to the predicate `checkThreshold` returns *true*, the corresponding request will be further verified using XDoS verification rules where the user's violation history is analyzed. The following Prolog rules demonstrate how to verify whether a user is 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 the threshold for frequency of service invocations. If the user's trust level is *high*, the XDoS verification rules only check the user's previous 3 sessions. If the user has 3 violations, his trust level is 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, the user's trust level is degraded to *low* or *permanentlyBlocked*, respectively. In all the above cases, if a query to the predicate `xdosVerify` evaluates to be *true*, the user's current session is 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 method `getHistorySessionStatus` to acquire information from the *State_Info* database, and invoking Java method `recordTrustLevel` to record a user's trust level as historical information in the *User_Info* database.

In order to demonstrate how to detect injection attacks using S-Wall, we utilize some simple examples of SQL injection attacks [15]. SQL injection is a technique used to exploit the vulnerabilities in web applications that communicate with databases [37]. 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 attack mostly occurs 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. Consider a web service that retrieves user information using the following query.

```
query = "SELECT * FROM user_records WHERE
userid = '" + uid + "'";
```

If `User1` is the user ID, the `WHERE` part of the query executed at the database will be "`userid = 'User1'`". Now a malicious user can obtain access to other users' records by tampering the parameter "`User1`" into "`User1' or 'x'='x`", so the resulting Boolean expression "`userid = 'User1' or 'x'='x'`" will evaluate to be *true*. When the query is executed, all records from the table *user_records* are returned. To prevent this type of attacks, the reasoning module may try to match the parameters of a web service invocation with predefined regular expressions [15]. For example, the following regular expression can be used to detect the aforementioned SQL injection attack.

```
([a-z0-9A-Z])*(\s*)(\')(\s*)(o|O)(r|R)
(\s*)(\') (\s*)([a-z0-9A-Z])*
```

The above regular expression specifies the string pattern "`' or '`", where `([a-z0-9A-Z])*` represents zero or more occurrence of alphanumeric characters; `(\s*)` represents zero or more occurrence of space characters; and `(\')` is the single quote.

Another 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 "`SELECT * FROM users WHERE userid = 'user1'`" is manipulated to "`SELECT * FROM users WHERE userid = 'user1'; DELETE FROM users;--j'`", the execution of the query results in data loss.

In our current implementation of the S-Wall, the reasoning module searches for string patterns such as "`' or '`", and concatenation of "`';`" and "`';--`" in the input strings. 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.

V. Case Study

In this section, we utilize a case study to demonstrate how S-Wall can be used to effectively defend against XML-based attacks. We developed a prototype S-Wall, and installed it on the same machine where a financial management service-oriented system was deployed. The financial management service-oriented system is the same one as we described earlier in Section III-A, which can be compromised by an attacker using a two-phased XML-based attack illustrated in the motivating example.

In the following, we first try to simulate the two involved XML-based attacks, namely XPath injection attack and a hybrid XDoS/overloaded payload attack. Then we demonstrate that, with S-Wall installed, the service-oriented system can effectively defend against these two major types of XML-based attacks

A. Simulation of XML-Based Attacks

As mentioned before, an XML settings file contains user authentication information and endpoint address of internal service SRS (refer to Fig. 1), which can be retrieved using XPath queries. To obtain the quote of a stock, a user can invoke the service operation *getQuote* defined in SIS, which requires three parameters, namely the ID of the user requesting the stock quote, the user password, and the ticker symbol on which the user wishes to receive a quote. A legitimate user request may look like `getQuote("afritz", "service1234", "GE")`, where `afritz` is a user name, `service1234` is a password, and `GE` is a ticker symbol. To authenticate user `afritz`, the user request is first transformed into the following XPath query for validation.

```
String Query =
  "//Service/users/user[loginID/text()=
  'afritz' and password/text() =
  'service1234']/usertype/text()"
```

After the user is authenticated, service operation *getQuote* invokes a remote service operation *getCurrentQuote* provided by GE Company, and returns the latest quote to the user.

Now suppose an attacker performs a two-phased XML-based attack on SMS. During the first phase of the attack, the attacker forms a crafted user request `getQuote("afritz","']| /* | /abc[def='","GE")`, which results in the following XPath query for user authentication.

```
String Query =
  "//Service/users/user[loginID/text()=
  'afritz' and password/text()=
  ''] | /* | /abc[def='']/usertype/text()"
```

In this query, the "`| /*`" notation instructs the XPath engine to return the entire XML file; thus, instead of authenticating the user, the attacker receives the entire contents of the XML file. The additional substring "`| /abc[def='']/usertype/text()`" in the query is simply used to ensure correct parsing when the request is transformed into an XPath query. Having the entire XML settings file, the attacker knows pertinent information about other services accessed by SIS, such as the endpoint address of SRS, credentials needed for accessing, timeout information, and other information stored in the XML file.

Once the endpoint address of the SRS service becomes available, the attacker can start to attack SRS. During the second phase of the attack, the attacker creates a hybrid XDoS/overloaded payload attack, and attempts to take down the SRS service. When the SRS service is down, all other services that rely on SRS, such as SIS and STS, will be dramatically affected, which may cause significant damage to the company and its clients. The reason why we chose to simulate a hybrid XDoS/overloaded payload attack instead of a pure overloaded payload attack is that, based on our experiments and experience, a pure overloaded payload attack that exhausts the XML parser does not appear to be an effective attack. By increasing the number of threads that perform overloaded payload attacks, the attack presents itself as a hybrid XDoS/overloaded payload attack, which can effectively exhaust the XML parser as well as the resources of the service being attacked. Note that the hybrid XDoS/overloaded payload attack differs from a pure XDoS attack because it does not require a large number of requests in order to exhaust the resources at the server side; thus, it will not be easily detected as an XDoS attack. For our case study, we developed an attacker client that can generate multiple threads to send out requests at a certain frequency. Each request is embedded with a large and complex header, which makes itself an overloaded payload attack. Since the SRS service requires user authentication that is processed prior to the service invocation, we utilize a handler authenticator to recursively explore the header for the credentials and continuously process the request if the credentials are correct. During the attack, we gathered statistics from normal clients attempting to use the service as designed, i.e., utilizing the SIS service to obtain stock information. The normal response time for a client is 100 milliseconds when SRS is not under attacks. Fig. 3 shows the results of the response time from four normal clients attempting to access the service simultaneously while the attacker client was running. Each normal client sends out a request, awaits a response, sleeps for a random amount of time, and then wakes up and makes a request again. As shown in the figure, the values of all observed response time are recorded over a period of 25 minutes, where the curve represents the average response time during this period. From the figure, we can see that after around 4 minutes, the average response time goes up to 10 seconds due to the hybrid XDoS/overloaded payload attack. After the service has been constantly attacked for around 25 minutes, the average response time exceeds 50 seconds, and the service eventually went down and became unable to service any further requests.

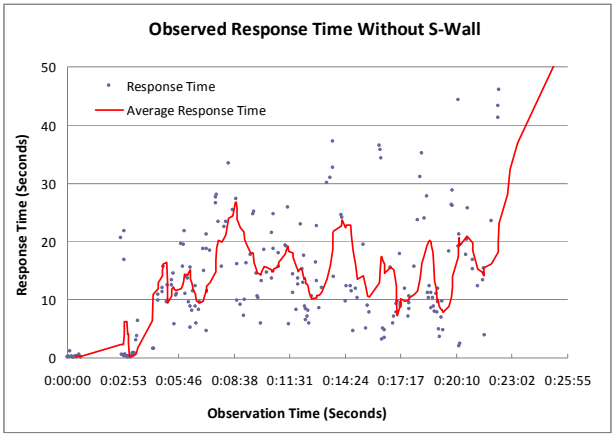

Figure 3. Observed response time without S-Wall

Fig. 4 categorizes the response time into 32 classes with an interval of 0.5 seconds, and records the number of requests in each category during the testing period. The curve in Fig. 4 represents the cumulative percentage of requests for a given response time category. From the figure, we can see that during the experiment, around 80% of the requests take over 3 seconds to get a response (i.e., around 20% of the requests take less than 3 seconds to get a response), and around 32% of the requests take over 15 seconds to get a response. Since the service under normal conditions takes less than one second to respond, the impact on the service is significant due to the hybrid XDoS/overload payload attack.

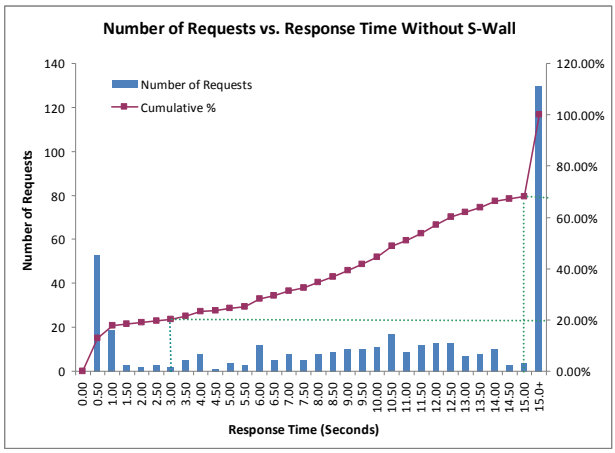

Figure 4. Number of requests vs. response time without S-Wall

B. *Defending Against XML-Based Attacks Using S-Wall*

When the S-Wall is installed, the service-oriented system can be protected from both the XPath attack and the hybrid XDoS/overloaded payload attack. Similar to the example of SQL injection attack detection described in Section IV-C, the reasoning module uses predefined regular expressions to check against XPath queries. Queries that match regular expressions of known XPath injection patterns are blocked and are not allowed to execute. Fig. 5 shows the log file for detection of an XPath injection attack against SIS.

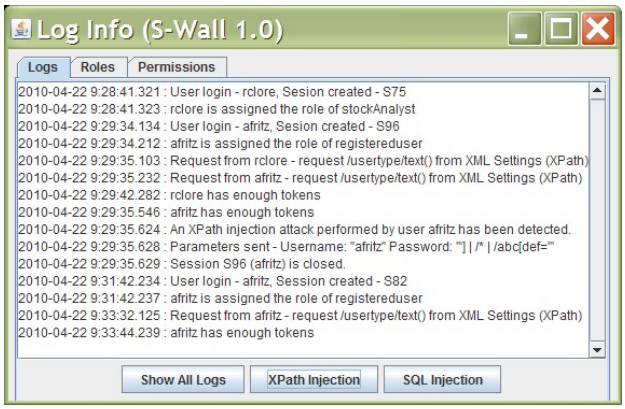

Figure 5. Log info for detecting XPath injection attacks

In this example, user "`afritz`" attempts to send a crafted user request `getQuote("afritz","'] |/*| /abc[def='","GE")` to SIS in order to obtain the location of the internal service. The S-Wall compares the XPath query with regular expressions defined in the *String Patterns* module (as shown in Fig. 2), and identifies the query as an XPath injection attack. Once the attack is recognized, the request is sent to the action module, where the users' session is terminated and the information related to the attack is recorded into the *User_Info* database. Note that since the location information of the internal service cannot be captured, the attacker will not be able to perform a hybrid XDoS/overloaded payload attack; thus, the two-phased attack can be effectively prevented.

In our next experiment, we assume that the attacker used some other technique (e.g., password cracking) to get the authentication information of a stock analyst. In this case, the attacker may still be able to perform a XDoS/overloaded payload attack on internal service SRS. In order to detect such a complex attack using S-Wall, we define D&V rules that can detect the nesting level and size of the elements in the message header. In configuring the S-Wall, the reasoning engine dynamically determines an acceptable nesting level based on the current user state and the firewall state. The following Prolog code demonstrates some sample detection rules that can be used to determine an appropriate nesting acceptable based on the state information.

```
nestingLimitBreach :-
  nestingLimit(S,SI,SEC,X),
  recordAttack(S,payload).
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=free, SEC=green, X>100.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=free, SEC=yellow, X>80.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=free, SEC=red, X>50.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=normal, SEC=green, X>80.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=normal,SEC=yellow, X>50.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=normal, SEC=red, X>20.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=busy, SEC=green, X>50.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=busy, SEC=yellow, X>20.
nestinglimit(S,SI,SEC,X) :-
  S=recomService, SI=busy, SEC=red, X>10.
recordAttack(S,A):-
  java_object('DataConnect',[],data),
  data <- recordAttackInstance(S,A).
```

In the above rules, `W` is the web service name, `SI` represents the state indication of the service, `SEC` represents the current firewall security level, and `X` is the number of nesting elements tolerated by the service. The predicate `nestingLimit` examines the current web service state indication, the firewall security level information, and the tolerance of nesting allowed for a particular header. A larger tolerance value implies less chance of blocking a legitimate request, but makes the web service more susceptible to attack. According to the above rules, there is a sliding scale of nesting allowed, with the maximum nesting of 100. The discrete scale

continues at 80, 50, 20, and 10. A deterioration in either the service state indication or the firewall security level causes a reduction in nesting tolerance. The nesting tolerance parameters can be easily set by an administrator to represent the expectations of a given service. The predicate `RecordAttack` is a java connector that can be invoked to record the attack instance when detected. This information is utilized in the verification rules to determine if the request was incidental or if there is a potential attack. Note that similar detection rules can be defined for the size of the nodes as well to detect overloaded payload attacks.

Once an attack is detected, the reasoning module needs to verify whether it is an effective attack using predefined verification rules, and based on the verification results, it may take actions accordingly. The verification process typically requires additional evidence stored as user information and state information. Notice that since a pure overloaded payload attack does not appear to be an effective attack, in our S-Wall implementation, we take actions only when a hybrid XDoS/overloaded payload attack has been verified. The following Prolog code gives some related sample verification rules.

```
hybridXdosPayloadVerify(U,T,S,SEC):-
  inspectHistory(U,T,S,SEC,A).
inspectHistory(U,T,S,SEC,A):-
  SEC = green, dataConnect(S,50,A),
  A > '20', T = normal,
  degradeSECLevel(SEC,yellow),
  degradeTrustLevel(U,low).
inspectHistory(U,T,S,SEC,A):-
  SEC = green, dataConnect(S,50,A),
  A > '20', T = low,
  degradeTrustLevel(U,permanentlyblocked).
inspectHistory(U,T,S,SEC,A):-
  SEC = yellow, dataConnect(S,100,A),
  A > '50' T = normal,
  degradeSECLevel(SEC,red).
dataConnect(S,X,A):-
  java_object('DataConnect',[],data),
  data <- getHistoryFirewallAttacks(S,X)
  returns A.
degradeSECLevel(U,T):-
  java_object('DataConnect',[],data),
  data <- recordSECLevel(SEC,X).
degradeTrustLevel(U,T) :-
  java_object('DataConnect',[],data),
  data <- recordTrustLevel(T,X).
```

In the above rules, `U` represents a user, `T` is the trust level of the user, `S` is a service, `SEC` is the current security level of the firewall, and `A` is the number of overloaded payload attacks detected during a predefined recent time interval. The predicate `dataConnect(S,X,A)` performs a lookup of the last `X` number of attacks performed against the service and checks whether a majority of them are overloaded payload attacks. The S-Wall also considers the current users' trust level prior to making a decision. For example, the first `inspectHistory` predicate verifies if the current `SEC` level is `green`, it then examines the last 50 requests as well as the users' trust level. If 20 of the requests were overloaded payload attacks and a normal user is performing the attack, both the users' trust level and the firewall's `SEC` level are degraded. Similarly, the second `inspectHistory` predicate verifies whether the user has a low rating stored in the *User_State* database. If so, the S-Wall blocks the user from further attempts without degrading the firewall security level. Note that when the security level increases, the firewall must perform more inspection of incoming requests and have stricter constraints. In order to avoid possible restriction of legitimate invocations at a heightened level of security, the firewall should be cautious in elevating its security level. To access the *User_Info* and *State_Info* database, it requires invoking three Java methods: `getHistoryFirewallAttacks` for acquiring historical firewall state information from the *State_Info* database; `recordSECLevel` for recording change of firewall status in the *State_Info* database; and `recordTrustLevel` for recording change of user status in the *User_Info* database.

After deploying the S-Wall, we were able to show significant improvements in responsiveness from the server. Running the hybrid XDoS/overloaded payload attack under the same conditions, we again collected the response time information of four normal clients, which simultaneously make requests to SRS. Fig. 6 shows the individual observed response times recorded over a period of 43 minutes, where the curve represents the average response time during this period.

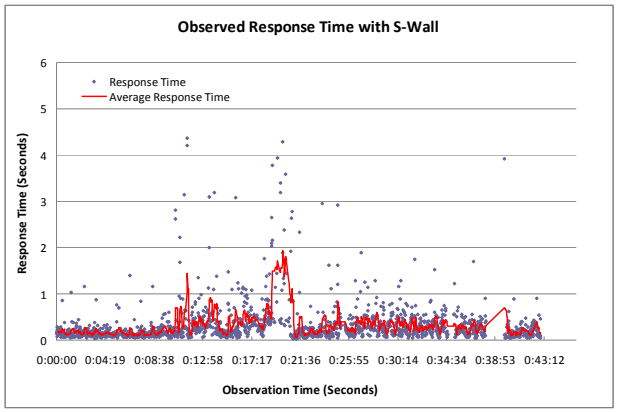

Figure 6. Observed response time with S-Wall

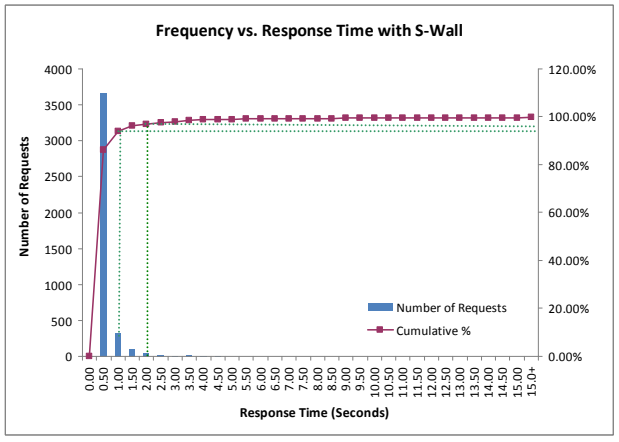

Figure 7. Number of requests vs. response time with S-Wall

The experimental results show that the average response time does not exceed 2 seconds during the 43

minutes testing period, and in most of the time, the average response is below 1 second. In Fig. 7, we categorize the response time into 32 classes with an interval of 0.5 second, and record the number of requests in each category. The curve represents the cumulative percentage of requests for a given response time category. From the figure, we can see that during the experiment, over 93% of the requests take 1 second or less, and over 97% of the requests take 2 seconds or less. Such results indicate the service is functioning properly with reasonable response times. Thus, with S-Wall installed, the SIS service was able to maintain a stable response time even when the SRS service was being attacked.

VI. Conclusions and Future Work

Service-oriented systems and cloud computing are becoming more and more popular due to their standardized protocols and techniques that enable the efficient integration of loosely coupled applications over the Internet. However, due to the open interface for service-oriented architecture and cloud computing, attacks on web services are more complicated than traditional web-based 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 security model called state-based XML firewall (S-Wall), which can be used to protect a service provider from various XML-based attacks. We developed a detailed design of the S-Wall security model, and implemented a prototype S-Wall to demonstrate the effectiveness of our approach. The experimental results show that S-Wall can efficiently and effectively detect and defend against various XML-based attacks. For future work, we will study new types of XML-based attacks and show how their corresponding D&V rules can be modularly constructed and integrated into our current system. We will also consider adopting agent-based technology to introduce more intelligence in S-Wall for detection and verification of more complicated XML-based attacks.

Acknowledgment

We thank the editor and all referees for the careful review of this paper. 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, University of Massachusetts Dartmouth.

References

[1] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, Prentice Hall PTR, Service-Oriented Computing Series, Aug. 2005.

[2] D. S. Linthicum, *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*, Addison-Wesley Professional, Oct. 2009.

[3] G. Raines, "Cloud computing and SOA," *Technical Report*, Service-Oriented Architecture (SOA) Series, The MITRE Corporation, Oct. 2009.

[4] Z. Jaroucheh, X. Liu, and S. Smith, "A model-driven approach to flexible multi-level customization of SaaS applications," in *Proc. 22nd Int. Conf. Software Engineering and Knowledge Engineering (SEKE'10)*, San Francisco, pp. 241-246, Jul. 2010.

[5] S. Mysore, "Securing web services - concepts, standards, and requirements," *White Paper*, Sun Microsystems, Inc., Santa Clara, CA, USA, Oct. 2003.

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

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

[8] E. Moradian and A. Håkansson, "Possible attacks on XML web services," *Int. J. Computer Science and Network Security (IJCSNS)*, vol. 6, no. 1B, pp. 154-170, Jan. 2006.

[9] P. Lindstrom, "Attacking and defending web services," *Technical Report*, Spire Security, LLC, Jan. 2004.

[10] S. Shah, *Hacking Web Services*, Charles River Media, Boston, Massachusetts, Aug. 2006.

[11] M. O'Neill, P. Hall-Baker, S. M. Cann, M. Shema, E. Simon, P. A. Watters, and A. White, *Web Services Security*, McGraw-Hill Osborne Media, Jan. 2003.

[12] A. Vorobiev, J. Han and N. Bekmamedova, "An ontology framework for managing security attacks and defenses in component based software systems," in *Proc. 19th Australian Conf. Software Engineering (ASWEC 2008)*, pp. 552-561, Mar. 2008.

[13] 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, Mar. 2006.

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

[15] A. Reddyreddy and H. Xu, "Securing service-oriented systems using state-based XML firewall," in *Proc. 20th Int. Conf. Software Engineering and Knowledge Engineering (SEKE'2008)*, Redwood City, San Francisco Bay, California, USA, pp. 512-518, Jul. 2008.

[16] H. Xu, M. Ayachit and A. Reddyreddy, "Formal modeling and analysis of XML firewall for service-oriented systems," *Int. J. Security and Networks (IJSN)*, vol. 3, no. 3, pp. 147-160, 2008.

[17] E. Bertino, L. Martino, F. Paci, and A. Squicciarini, *Security for Web Services and Service-Oriented Architectures*, Springer, 2009.

[18] E. B. Fernandez, "Two patterns for web services security," in *Proc. 2004 Int. Symp. Web Services and Applications (ISWS'04)*, Las Vegas, Nevada, 2004.

[19] M. Holtkamp, "The role of XML firewalls for web services," in *Proc. 1st Twente Student Conference IT*, Jun. 2004.

[20] M. Cremonini, S. Vimercati, E. Damiani, and P. Samarati, "An XML-based approach to combine firewalls and web services security specifications", in *Proc. 2003 ACM Workshop XML Security*, Fairfax, Virginia, pp. 69-78, Oct. 2003.

[21] R. Bebawy, H. Sabry, S. El-Kassas, Y. Hanna, and Y. Youssef, "Nedgty: web services firewall," in *Proc. IEEE Int. Conf. Web Services (ICWS'05)*, pp. 597-601, 2005.

[22] Forum, *Forum XWall*, Forum Systems, Inc., Retrieved on Feb. 18, 2008, from http://forumsys.com/products_xwall.htm.

[23] S. Northcutt and J. Novak, *Network Intrusion Detection*, 3rd Edition, Sams, Sept. 2002.
[24] R. G. Bace, *Intrusion Detection*, Macmillan Technical Publishing, Indianapolis, IN, USA, 2000.
[25] Y. Mai, R. Upadrashta, and X. Su, "J-Honeypot: a Java-based network deception tool with monitoring and intrusion detection," in *Proc. Int. Conf. Information Technology: Coding and Computing (ITCC 2004)*, Las Vegas, NV, USA, pp. 804-808, Apr. 2004.
[26] W. Zhang, R. Rao, G. Cao, and G. Kesidis, "Secure routing in ad hoc networks and a related intrusion detection problem," in *Proc. IEEE Military Communications Conference (MILCOM)*, Oct. 2003.
[27] K. Rao, A. Pal, and M. R. Patra, "A service oriented architectural design for building intrusion detection systems," *Int. J. Recent Trends in Engineering*, vol. 1, no. 2, pp. 11-14, May 2009.
[28] J. McGibneya, N. Schmidtb, and A. Patelb, "A service-centric model for intrusion detection in next-generation networks," *Computer Standards & Interfaces*, vol. 27, no. 5, pp. 513-520, Jun. 2005.
[29] B. Zhou, Q. Shi, and M. Merabti, "A framework for intrusion detection in heterogeneous environments," in *Proc. 4th IEEE Consumer Communications and Networking Conference (CCNC 2006)*, pp. 1244-1248, Jan. 8-10, 2006.
[30] H. Feinstein, R. Sandhu, E. Coyne, and C. Youman, "Role-based access control models," in *Proc. IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.
[31] G. Zhang and M. Parashar, "Context-aware dynamic access control for pervasive applications," in *Proc. Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004)*, Western Multi-Conference (WMC), San Diego, CA, USA, Jan. 2004.
[32] H. Xu, S. M. Shatz, and C. K. Bates, "A framework for agent-based trust management in online auctions," in *Proc. 5th Int. Conf. Information Technology: New Generations (ITNG 2008)*, Las Vegas, Nevada, USA, pp. 149-155, Apr. 7-9, 2008.
[33] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224-274, Aug. 2001.
[34] M. Becker, "Cassandra: flexible trust management and its application to electronic health records," *Ph.D. Thesis*, University of Cambridge, Oct. 2005.
[35] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: a logic-based approach to distributed authorization," *ACM Trans. Information and System Security*, vol. 6, no. 1, pp. 128-171, Feb. 2003.
[36] J. DeTreville, "Binder, a logic-based security language," in *Proc. 2002 IEEE Symp. Security and Privacy*, IEEE Computer Society Press, pp. 105-113, May 2002.
[37] C. Anley, "Advanced SQL injection in SQL server applications," *White Paper*, Next Generation Security Software Ltd., Jan. 2002.

Haiping Xu received the B.S. degree in Electrical Engineering from Zhejiang University, Hangzhou, China, in 1989, the M.S. degree in Computer Science from Wright State University, Dayton, OH, in 1998, and the Ph.D. degree in Computer Science from the University of Illinois at Chicago, IL, in 2003.

Prior to 1996, he successively worked with Shen-Yan Systems Technology, Inc. and Hewlett-Packard Co., as a Software Engineer, in Beijing, China. Since 2003, he has been with the University of Massachusetts Dartmouth, where he is currently an Associate Professor at the Computer and Information Science Department, and a Director of the Concurrent Software Engineering Laboratory (CSEL). He has published about 50 research papers including over 20 peer-reviewed journal publications. He has supervised over 30 M.S. theses and M.S. projects at the University of Massachusetts Dartmouth, and co-supervised 2 Ph.D. dissertations. His research has been supported by the National Science Foundation (NSF) and the U.S. Marine Corps. His research interests include distributed software engineering, formal methods, Internet security, multi-agent systems, electronic commerce, service-oriented systems, and cloud computing.

Dr. Xu is a senior member of both the IEEE and the Association of Computing Machinery (ACM). He has served as a program committee Co-Chair for the International Conference on Software Engineering Theory and Practice (SETP), and a program committee member for over 50 international conferences. He is a recipient of the Outstanding Ph.D. Thesis Award in 2004, and has been included in the 11th Edition of Who's Who Among America's Teachers, 2006.

Abhinay Reddyreddy received the B.Tech degree in Computer Science and Engineering from the Jawaharlal Nehru Technological University, India, in 2005, and the M.S. degree in Computer Science from the University of Massachusetts Dartmouth, North Dartmouth, MA, in 2008. He is currently a software engineer in Boris FX, Inc. in Marlborough, MA. His research interests include software engineering, web services security, and formal methods for specification and analysis of concurrent and distributed software, especially the application of Petri net-based models.

Daniel F. Fitch received the B.S. degree in Computer Science from the Bridgewater State University in 2008. He is currently a graduate student in the Computer and Information Science Department at the University of Massachusetts Dartmouth. His research interests include software engineering, web services security, and formal methods for specification and analysis of reliability and fault tolerance model in cloud computing.