A PETRI NET BASED XML FIREWALL SECURITY MODEL FOR WEB SERVICES INVOCATION

Mihir M. Ayachit and Haiping Xu
Computer and Information Science Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747
Email: {g_mayachit, hxu}@umassd.edu

ABSTRACT
An XML firewall differs from a conventional firewall because its major task is to control access to web services rather than to filter untrusted addresses. An XML firewall can effectively protect web services from being attacked by inspecting a complete XML message including its head and data segments, and rejecting unauthorized web services invocation. In this paper, we propose a formal XML firewall security model using role-based access control (RBAC). Our proposed model supports user authentication and user authorization according to information stored in a user database and a policy database associated with an XML firewall. The formal model is designed compositionally using Petri nets, which can serve as a high-level design for XML firewall implementation. The key components of our compositional security model are the application model and the XML firewall model. To illustrate the advantages of our formal approach, we use an existing Petri net tool to verify some key properties of our model, such as boundedness and liveness.

KEYWORDS
XML firewall, web services, role-based access control (RBAC), Petri net model, formal verification

1. Introduction

Web services are Internet-based software components that support open, XML-based standards and communication protocols. As more businesses deploy web services into applications that dynamically interact with other applications and data sources, the issue of how to secure them from intruders and other possible threats becomes more important [1]. Security problems in web services are severe because the Internet is an insecure and untrustable public network infrastructure, where the information available to be accessed over the Internet has different levels of business confidentiality. Furthermore, a service consumer may invoke web services using false identity, or corrupt the services by attacking the service providers, for example, using a denial of service attack. Thus, security consideration becomes very critical for the successful deployment of web services applications.

Conventional firewalls have been designed as a major component to protect a network or a server from being attacked. However, they may provide no security at all for web services. This is because web services normally use the SOAP protocol over HTTP, whose port is typically not blocked by conventional firewalls. To protect web services from being attacked, we develop a general framework, called XML firewall security model, which enforces access restrictions for web services invocation. In our model, the access to web services is only granted to those users who are authenticated and authorized to have access to the services. The model is formally defined using the Petri net formalism, which is a mature formalism with existing theory and tool support [2]. There are two types of components in the XML firewall security model, namely, the application model and the XML firewall model. In the XML firewall model, we adopt the role-based access control (RBAC) mechanism to effectively deploy user authorization and access rights. The RBAC model has been proposed as one of the most attractive solutions to providing security features in different distributed computing infrastructure [3]. 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 ensures that only authorized users are given access to certain data or resources. Most of the RBAC models follow the same basic structure of subject, role and privilege. However, in a more sophisticated role-based access control model, access decisions for an application will depend on the combination of the required credentials of users and the context and state of the system, as well as other factors such as relationship, time and location [4]. The RBAC mechanism we use in our model depends on not only the user's identity, but also the current state of the system. In our XML firewall, we can define certain policy rules that specify the users' access to the web services based on the system state. Thus, our XML firewall model is *stateful*.

There is very little work done in the past on how to protect web service providers from being attacked. Previous work on protecting web services from unauthorized access emphasized on developing pattern-

based language for XML firewall [5]. Fernandez and his colleagues classified firewalls into three categories, namely, packet filter firewall, proxy-based firewall, and stateful firewall [6]. They proposed two patterns for XML firewall, which are security assertion coordination pattern using RBAC for access to distributed resources, and filter pattern for filtering XML messages or documents according to institution policies. Instead of proposing pattern languages for stateless XML firewall, we design a stateful XML firewall protected system that may assign permissions to various roles according to the current system. Furthermore, since XML firewalls are critical systems for businesses, to ensure the correctness of the system design, we develop a formal model using Petri nets, and demonstrate how existing Petri net tools can be used to verify the key properties of our model.

Currently, there are also a few XML firewall products available in the market to secure web services developed by leading companies. For example, the Forum Systems Company has an XML security appliance that is a combination of hardware and software and resides in front of servers that contain sensitive XML tagged information [7]. The appliance encrypts XML fields in real time, as the data goes into the server. It then decrypts it when the data exits the server. Although such XML firewall implementations can help to protect web services, their functionalities are still very limited. For example, they are usually not state-based, so they cannot protect web services from certain threats such as a denial of service attack. In this paper, we propose a general solution to implementing XML firewalls based on a Petri net based XML firewall security model, which is formally defined and supports formal verification as we did in our previous work [8]. Meanwhile, our formal model can serve as a high-level design for XML firewall implementation, and may provide a potential solution to automated software development as illustrated in [9].

The rest of the paper is organized as follows. Section 2 presents an architectural design of XML firewall protected systems. Section 3 introduces the compositional Petri net based XML firewall security model, including the application model and the XML firewall model. Section 4 performs some formal analysis of the Petri net models using an existing Petri net tool. Section 5 gives the conclusions and future work.

2. Architectural Design of XML Firewall

To deal with security issues in web services invocation, we build an XML firewall security model to protect web services from threats in an unsecured environment. Such threats include unauthorized access and access without sufficient permissions. Our approach focuses on building the XML firewall model that coordinates authentication and access rights. The proposed model for XML firewalls can filter XML messages according to policies enforced in a policy base associated with each XML firewall.

The XML based firewall security model consists of three major components: applications, XML firewalls and web services. The architecture of an XML firewall protected service-oriented system is illustrated in Figure 1. As shown in the figure, a user interacts with the application through the user interface. The application logic is the business logic inside the application, which varies from application to application. The application logic processes the requests from the user and initiates service calls. A service call can be an invocation of a single web service or a group of web services. The request from the application is checked by the XML firewall for authenticity and access limitations depending on the system state. If the request is valid, the XML firewall will pass the request to the corresponding web service; otherwise, the request is rejected. The administrator of the XML firewall can change the policies of the firewall at runtime. Each web service has its own logic to process the corresponding method request and returns the result to the XML firewall. Upon receiving the results from the web services, the XML firewall passes the results back to the application. When the application receives the results from the XML firewall, the application logic processes these results and may send appropriate messages to the user through its user interface.

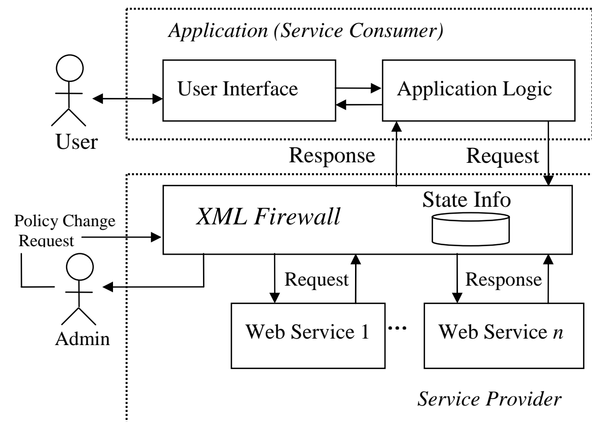

Figure 1. Protecting service provider using stateful XML firewall

Figure 2 is the refinement of the XML firewall, which describes the important components inside an XML firewall model. When a user starts the application, he first logs into the application. Then the user's access requests are processed by the computational logic. Based on the user's requests, the computational logic initiates the needed service calls. The service call with the user's information is intercepted by the XML firewall for authentication and authorization. The user is authenticated by checking against the *UserInfo* database, i.e., the *UserInfoDB* as shown in Figure 2. If the user's identification is valid, he is assigned a role from the *Role* database, i.e., the *RoleDB*; otherwise, an *access denied* message is sent to the application. The role assignment is based on the current state of the user as well as the state

of the system, which is determined by the status of incoming message and the information stored in the *State* database, i.e., the *StateDB*. After the role assignment is done, a user space is created by using policies from the *Policy* database (i.e., the *PolicyDB*), which contains the access permissions of the user. The *user space* is then compared with the service request to determine whether the incoming request from the user has permissions to invoke the web service. If the user has the permissions, then the request is passed to the corresponding web service; otherwise, an *access denied* message is sent to the application. Upon receiving a request from the XML firewall, the web service process the request and returns the result to the XML firewall, which is then passed back to the computational logic in the application.

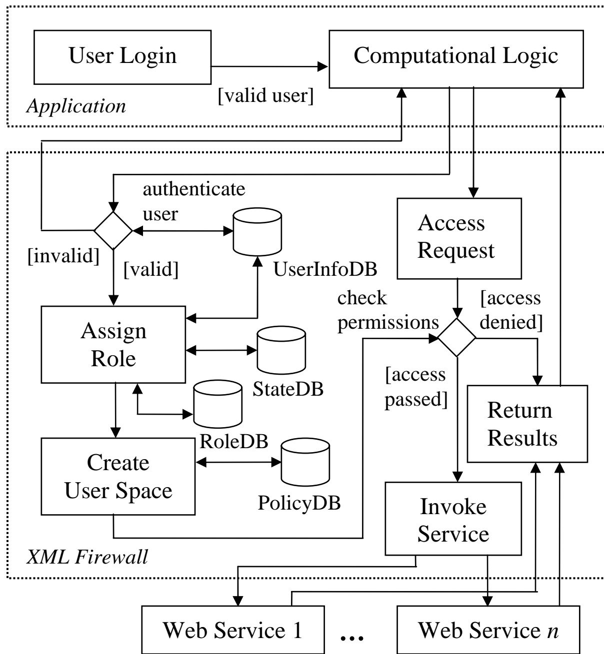

Figure 2. XML firewall architecture

3. Compositional Petri Net Models of XML Firewall

Petri nets are a graphical and mathematical modeling tool applicable to many systems [2]. A Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition. In a Petri net model, tokens are used to specify information or conditions in the places. For an ordinary Petri net, when there is at least one token in every input place of a transition, the transition is enabled. An enabled transition may fire by removing one token from every input place, and depositing one token in each output place of the transition. In this section, we develop a compositional XML firewall security model for web services invocation using Petri nets. As mentioned previously, we design our XML firewall protected service architecture using modular design with the basic modules, i.e., the application model and the XML firewall model, where the interfaces between these modules are well defined.

3.1 Application Model

Figure 3 shows the Petri net model of an application that invokes two web services concurrently. In the application model, we assume that a user can log into the application by providing his username and password. Once a user provides his username and password, a token is placed in the *Login_Request* place. The username and password are then received by firing *Get_Login_Request* transition and a token is deposited into the *Username_Password* place. The *Check_User_DB* transition is fired to check the validity of the username and password according to the information stored in the *User_DB* place, which is a database that stores details of all registered users, for example, the user's contact information. If the username and password check is valid, the *Get_User_Details* transition is fired and a token is placed in the *User_Details* place. At the same time, a token is deposited into the *Ready_To_Accept_Request* place to indicate that now a user access request can be processed. It should be noted that a user could make a request to the application only if he is authenticated by the application. If the user fails the authentication check, then a token is placed in the *Failure* place by firing the *Not_Valid* transition. In this case, the transition *Access_Denied* can fire and a token will be returned to the *Login_Request* place. The token placed in the *User_Access_Request* place represents a request from the user. The user request is accepted by firing the *Access_Request* transition. Note that the *Access_Request* transition can fire only if there is a token in the *Ready_To_Accept_Request* place. As a result of firing the *Access_Request* transition, a token is deposited into the *Dispatch_Request* place. If the request is a logout request, then the *Logout* transition will fire. If the *Logout* transition fires, a token is taken out of the *Ready_To_Accept_Request* place and *User_Details* place, and a new token is returned back to initial place *Login_Request*. Since there is no token in the *Ready_To_Accept_Request place* now, a user must login again before he can make further access requests.

If the request made by the user is an access request, the *Create_Request* transition can fire, and a token will be deposited into the *Request_Details* place. A token in the *Request_Details* place contains the information retrieved from the *User_Details* place combined with the information from the incoming user request. The token in place *Request_Details* enables the *Computational_Logic* transition, which represents the business logic of the application. The *Computational_Logic* transition is defined as an abstract transition (denoted as shaded rectangle in Figure 3), which is a unit of module that can be refined later on. When the transition *Computational_Logic* fires, the application applies its business logic to the incoming request and generates requests for web services invocation. To illustrate

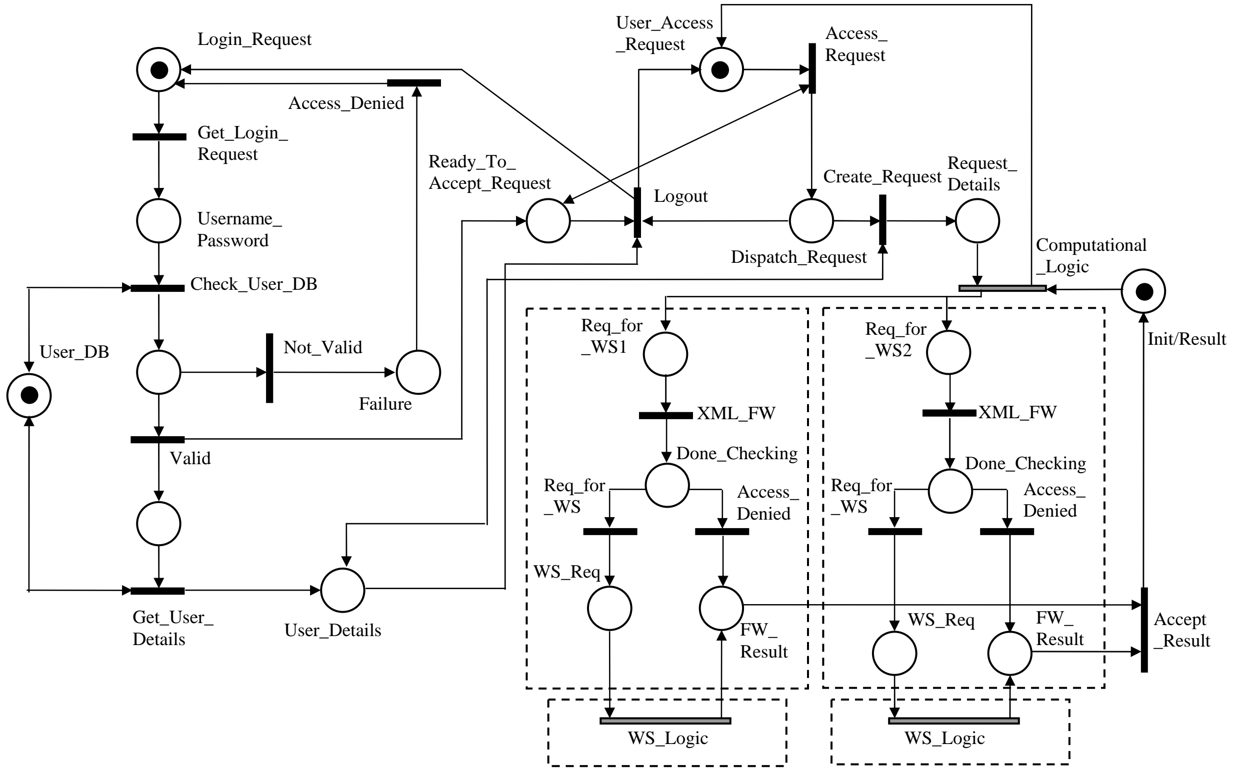

Figure 3. Petri net model of an application that invokes two web services concurrently

concurrent invocations of two web services, the application model includes two web services that are protected by a XML firewall respectively. To simplify matters, we assume that the user has to wait for the results of both of the requests to be processed before any further requests can be made. Notice that the XML firewall model shown in the figure (in a dashed line box) can be used to secure a single or a group of web services. We will refine the XML firewall in Section 3.2. The goal of the XML firewall is to perform the authentication and authorization verification of incoming requests from the application. Hence, when the application accepts the request, the XML firewall performs the authentication verifications and checks the access rights. If the user is an authorized user, and if he has the necessary permissions to the web service that is requested, then the web service will be invoked. This logic is represented in the Figure 3 as the *XML_FW* transition. By firing the transition *XML_FW*, a token is deposited in place *Done_Checking* for further processing. If the user request is authentic and the user has all the necessary permissions, the transition *Req_for_WS* can fire. When the transition fires, a token representing this request will be deposited into place *WS_Req*, and enables the *WS_Logic* transition. The transition *WS_Logic* is defined as an abstract transition that represents the corresponding web service logic. After processing the web service, a token representing the result is deposited into the *FW_Result* place. On the other hand, if the web service access is denied, the *Access_Denied* transition fires, and a token representing an *access denied* message is placed in the *FW_Result* place.

Now the *Accept_Result* transition in the application can fire if we have a token in the *FW_Result* place in both of the XML firewalls. Once the result is accepted, a token is deposited into the *Init/Result* place, which can be used by *Computational_Logic* transition for further processing. After the *Computational_Logic* transition fires, a token is returned to the *User_Access_Request* place, which enables the next user access request.

3.2 XML Firewall Model

The XML firewall module in Figure 3 (displayed inside the dashed line box) can be refined as shown in Figure 4. To make the Petri net model self-contained, we have shown an abstraction of the application model with two places and two transitions. In this model, we also include an abstract web service module that is denoted by the abstract transition *WS_Logic*.

As we discussed earlier, the computational logic in the application handles all the incoming requests coming from the user and invokes the corresponding web services. When the *Computational_Logic* generates a web service request, a token is placed into the *WS_Request* place indicating a method call. The *Check_If_Existing* transition can fire in order to check if the user is an existing user or a new one, where the user who made the request is checked for identity. If the user is not found in the *UserInfo_ DB,* then the user is recognized as a first time user and the *First_Time_User* transition can fire. For each first time user, the *Perform_Background_Check*

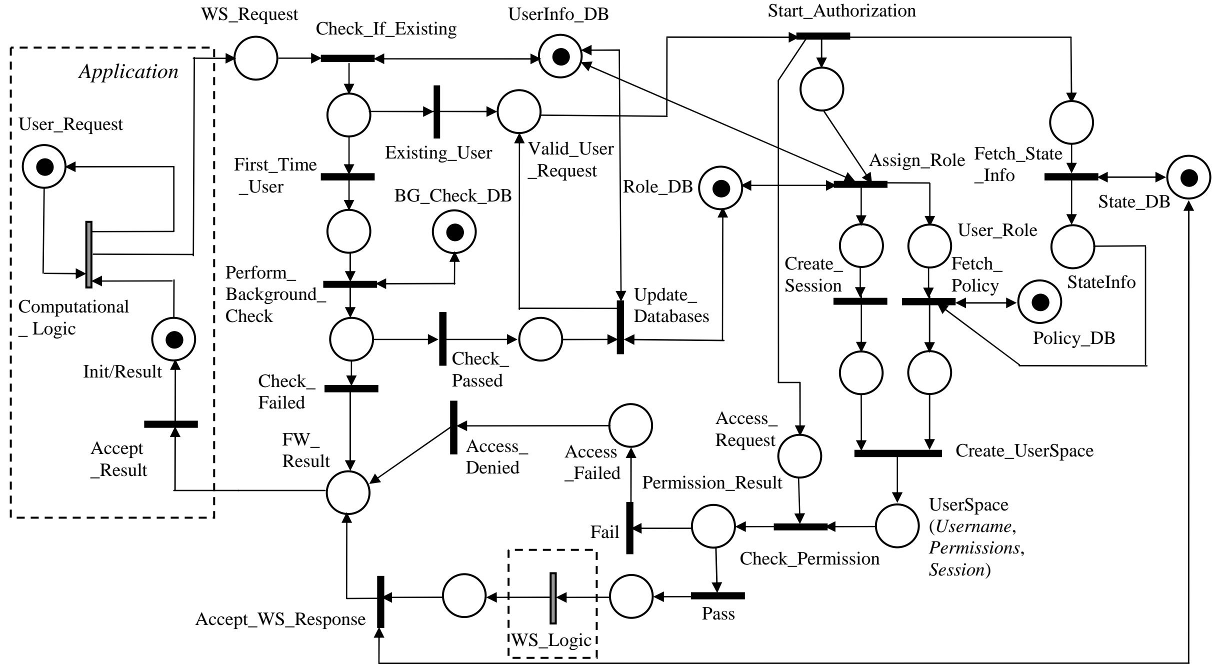

Figure 4. Petri net model of an XML firewall with one application and one web service

transition can fire and a background check is performed using information stored in *BG_Check_DB*. A user becomes a valid member if the background check is passed. As a result of valid authentication, the *Update_Databases* transition is fired to update the *UserInfo_DB* and *Role_DB*. Meanwhile, a token is deposited into the *Valid_User_Request* place indicating a valid user request. If the authentication fails, the *Check_Failed* transition will fire and a token indicating access denied is placed in the *FW_Result* place.

The user is identified as a regular user if his user profile exists in the *UserInfo_DB* database. For a regular user, the *Existing_User* transition can fire and a token is deposited into the *Valid_User_Request* place. Once the token is deposited into the *Valid_User_Request* place, the authorization process starts by firing the *Start_Authorization* transition. The state information for the incoming request is generated by firing the *Fetch_State_Info* transition, which uses state information that is already stored in the *State_DB*. Since the incoming request may hold state information itself (e.g., the time of the request), the state of the incoming request is computed using *State_DB* as well as the status of the incoming request. After the state information is generated, a token indicating current state of the request is placed in the *State_Info* place. Then, the *Assign_Role* transition can fire to assign the roles to the user using information stored in the *Role_DB*. In addition, a user session is created by firing the *Create_Session* transition. The user session defines the period of time during which, a user can interact with an application. The user session begins when the user starts to access the web services and ends when the user finishes the web services invocation. If the session expires during the invocation, the *WS_Logic* transition returns a *timeout* result back to the XML firewall. The next task is to fetch a policy from the *Policy_DB*. The *Fetch_Policy* transition can fire when there is a token in the *User_Role* place and the *State_Info* place. A policy is fetched from the *Policy_DB* based on the user role and current state of the system. After a policy is fetched and a session is created, a *user space* is created that contains the user information, permissions and the session information. A token representing a user space will be deposited into the *UserSpace* place.

A token in the *Access_Request* place represents a web service invocation request. The *Check_Permission* transition can fire to check the *Access_Request* with the *User_Space* to determine its access permissions. After the checking, a token representing the result will be deposited into the place *Permission_Result*. If the user has the needed permissions, then the *Pass* transition fires. After the web service request is processed (i.e., the firing of the *WS_Logic* transition), a token representing the result of the web service invocation is passed to the XML firewall. This token enables the *Accept_WS_Response* transition. The result from the web service also updates information in the *State_DB*. On the other hand, if the user does not have sufficient permissions to invoke a web service, the *Fail* transition fires, and a token representing access denial is placed in the *Access_Failed* place. When the transition *Access_Denied* fires, a token is deposited into the *FW_Result* place, which indicates the web service access is denied. From the above model, we can see that the *FW_Result* place may hold two types of tokens: one representing an *access denied* message, and another one representing the result from web service invocation. With

a token in the *FW_Result* place, the transition *Accept_Result* defined in the abstract application model can fire. As a result, a token will be deposited into the *Init/Result* place, and the *Computational_Logic* transition will decide the next step of the actions. Whenever the *Computational_Logic* transition fires, a new token will be returned to the initial place *User_Request* to allow further user requests.

4. Analysis of XML Firewall Model

One of the advantages of using Petri nets to model XML firewall protected systems is its support for formal analysis using existing Petri net analysis tools. In this section, we show how to use the INA (Integrated Net Analyzer) tool [10] to analyze some key properties of our model. The INA tool is a program that can be used to analyze a Petri net model for its general properties, for example, the safety and liveness property. The INA tool is an interactive analysis tool that incorporates a large number of methods for analysis of Petri nets. These methods include analysis of structural properties such as structural boundedness, T- and P- invariant analysis and behavioral properties, such as boundedness, safety, liveness, and deadlock-freeness. To verify the correctness of our XML firewall models, we utilize some key definitions for Petri net behavior properties as adapted from [2].

Definition 4.1 Reachability: In a Petri net N with initial marking M_0, denoted as (N, M_0), a marking M_n is said to be *reachable* from a marking M_0 if there exists a sequence of firings that transforms M_0 to M_n. A firing or occurrence sequence is denoted by $s = M_0\ t_1\ M_1\ t_2\ M_2\ \ldots\ t_n\ M_n$ or simply $s = t_1\ t_2\ \ldots\ t_n$. In this case, M_n is reachable from M_0 by s and we write $M_0\ [s > M_n$.

Definition 4.2 Boundedness: A Petri net (N, M_0), is said to be *k-bounded* or simply *bounded* if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0. A Petri net (N, M_0) is said to be safe if it is 1-bounded.

Definition 4.3 Liveness: A Petri net (N, M_0), is said to be *live* if for any marking M that is reachable from M_0, it is possible to ultimately fire any transition of the net by progressing some further firing sequence.

Definition 4.4 Reversibility: A Petri net (N, M_0) is said to be *reversible* if, for each marking M that is reachable from the initial marking M_0, M_0 is reachable from M.

We first use our net model in Figure 3 as an input to the INA tool. The INA tool produces the following results:

```
Deciding structural boundedness
The net is structurally bounded.
The net is bounded.

Computation of the reachability graph
States generated: 238
The net has no dead transitions at the initial
marking.
The net has no dead reachable states.
The net is safe.

Livenesstest:
Computing the strongly connected components
The net is live.
The net is live, if dead transitions are
ignored.
The net is live and safe.
The net is reversible (resetable).
```

The analysis result shows that our net model is live. Thus, for any marking M that is reachable from M_0, it is possible to ultimately fire any transition of the net. As a result, as long as there are valid user requests with the needed permissions, the *WS_Logic* transition can fire eventually.

The result also shows that our model is bounded and safe. This means that each place in the net may contain at most one token at any time. For example, only one token is allowed to be deposited into the place *Dispatch_Request* at any time. This model works properly when user requests are handled sequentially. However, to handle multiple user access requests at the same time, we need to revise our net model as follows: we first remove the arc from the *Computational Logic* to the *User_Access_Request* place, and then we make the arc between the *User_Access_Request* place and the *Access_Request* transition bidirectional. As a result of these changes, there can be multiple tokens in the *Dispatch_Request* place, which shows that multiple user access requests can be handled concurrently. Finally, the analysis tells us that our net model is reversible. This indicates that the initial marking M_0 can be reproduced (by Definition 4.4). Since the initial marking M_0 represents that there are no web service requests being processed at the net. The reversibility property proves that every web service request can be processed successfully.

Now using our net model defined in Figure 4 as an input to the INA tool, it produces the following results:

```
Deciding structural boundedness
The net is structurally bounded.
The net is bounded.

Computation of the reachability graph
States generated: 34
The net has no dead transitions at the initial
marking.
The net has no dead reachable states.
The net is safe.

Computing the strongly connected components
The net is live.
The net is live, if dead transitions are
ignored.
The net is live and safe.
The net is reversible (resetable).
```

By showing that our XML firewall net model is live, we prove that under all circumstances, it is possible to

eventually fire any transition in the net. For example, the transition *Start_Authorization* is proved to be live. However, in our model, every incoming web service request from the application is checked for authentication first. If the authentication is passed, then the request is further processed for authorization. On the other hand, if the authentication check fails, our net model will send an *access denied* message to the application without doing any authorization operations. This indicates that the *Start_Authorization* transition can fire *only if* the authentication has been passed (i.e., the user is an existing user or the background check has been passed for a first-time user). Since we cannot guarantee the existence of a valid (authenticated) user, we cannot guarantee that the transition *Start_Authorization* can fire eventually. This conflicts with the analysis result, which says that the transition *Start_Authorization* is live. By looking into our net model, we observe that we have simplified our net model with ordinary tokens or black tokens. Since the black tokens do not hold sufficient information to indicate the success or failure of a background check, it is always possible to fire the transition *Check_Passed* in Figure 4. As a consequence, the transition *Start_Authorization* can ultimately be fired. To solve this problem, we can use colored tokens instead of ordinary tokens to represent the background check results, and attach guards for transitions. In this case, both the transition *Check_Passed* and *Start_Authorization* cannot fire if the background check fails.

The analysis result also shows that our net model is safe and bounded. Since we have shown only one application in our net model, we can expect that there can be at most one token in the *WS_Request* place at any time. However, when there are multiple applications that invoke multiple web services at the same time, more than one token can be deposited into the *WS_Request* place, and the net model becomes no longer safe. In addition, the analysis result shows that the model is reversible, which indicates that after the invocation of a web service, the system can successfully return to its initial state.

Notice that the Petri net models we have developed in this paper are compositional. This means we can easily develop a Petri net model that consists of multiple applications, multiple firewalls, and multiple web services. Since both of the application model and the XML firewall model have been proved to be live, it is easy to prove that a compositional model with multiple applications, firewalls and web services is also live.

5. Conclusions and Future Work

We introduced a Petri net based XML firewall security model that supports secured web services invocation. Our approach adopts the role-based access control (RBAC) mechanism, so user roles and permissions for web services invocation can be assigned dynamically. Our approach is *stateful*, where user permissions are assigned according to system and user's state information including user's previous web service invocation history. Therefore, our approach may effectively protect web services from certain type of threats, for example, the denial of service attack. To illustrate the advantages of our formal approach, we use the INA tool to verify that our Petri net models are live and safe. In our future work, we plan to refine our Petri net models and build a prototype of stateful XML firewall protected service-oriented system, and show that our approach is feasible and effective.

References

[1] S. Mysore, Securing web services – concepts, standards, and requirements, *White paper*, Sun Microsystems, October 2003.

[2] T. Murata, Petri nets: properties, analysis and applications, *Proce of the IEEE*, *77*(4): 541-580, April 1989.

[3] H. Feinstein R. Sandhu, E. Coyne and C. Youman, Role-based access control models, *IEEE Computer*, *29*(2):38–47, 1996.

[4] Guangsen Zhang, Manish Parashar, Context-aware dynamic access control for pervasive applications, *Proc. of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004)*, Western Multi-Conference (WMC), San Diego, CA, USA, 2004.

[5] E. B. Fernandez, Two patterns for web services security, *Proc. of the 2004 Intl. Sym. on Web Services and Applications (ISWS'04)*, Las Vegas, NV, 2004.

[6] E. B. Fernandez, M. M. Larrondo-Petrie, N. Seliya, N. Delessy-Gassant, and M. Schumacher, A pattern language for firewalls, In M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad (Eds.), *Security Patterns*, Wiley 2005.

[7] D. Allen, *Forum Systems' XWall Web Services Firewal*, Retrieved on February 29, 2006, from http://www.networkmagazine.com/shared/article/showArticle.jhtml?articleId=18900090

[8] H. Xu and S. M. Shatz, A framework for model-based design of agent-oriented software, *IEEE Transactions on Software Engineering (IEEE TSE)*, January 2003, Vol. 29, No. 1, pp. 15-30.

[9] H. Xu and S. M. Shatz, ADK: an agent development kit based on a formal model for multi-agent systems, *Journal of Automated Software Engineering (AUSE)*, October 2003, Vol. 10, No. 4, pp. 337-365.

[10] S. Roch and P. H. Starke, *INA: Integrated Net Analyzer*, Version 2.2, Humboldt-Universitat zu Berlin, Institut für Informatik, April 1999.