Defending Against Denial of Web Services Using Sessions

Jun Wang
C&C Research Laboratories, NEC Europe Ltd.
53757 Sankt Augustin, Germany
wang@ccrl-nece.de

***Abstract*—With Web services being accepted and deployed in both research and industrial areas, the security related issues become important. Denial of Web Services (DoWS), as an easily constructed attack approach, is a potential threat for the deployment of Web services, especially on a wide area network. In this paper, we introduce the session based ideas into the access of Web services and develop a fair share based filtering algorithm to improve the resistance capability of a Web service to DoWS. The implementation of our framework on the Java platform is presented. The preliminary experiment and its result show that our framework can improve the resistance capability of a Web service to DoWS without modification to the existing client side programming models. The extra overload introduced by our framework to the server side is low.**

I. Introduction

According to [1], "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Service Definition Language (WSDL) [2]). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." Web services, as a new building block in distributed computing environments, are beginning to be deployed by more and more organizations for their Enterprise Application Integration (EAI) in their private LAN or in their VPN. However, most organizations are still hesitating to deploy Web services in an open and public WAN. One important reason is the security of Web services. Currently, the Web services community has produced a series of specifications in order to enhance the security of Web services from basic WS-Security [3] that aims at end-to-end security to WS-Trust [4], WS-SecureConversation [5] and WS-SecurePolicy [6] that aim at securing the exchange of multiple messages in multiple trust domains with different local policies. All of the existing specifications do not touch Denial of Web Services (DoWS) though.

From the definition of Web services above, we can see that a typical Web service request is an XML encoded SOAP message transferred over HTTP. Each Web services hosting environment has therefore to perform some time-consuming XML processing steps such as serialization, deserialization as well as parsing in order to generate or understand a service request. DoWS attacks might exploit this property and take advantage of it. In this context a DoWS attack will degrade or even deny the normal access to a Web service by flooding it with useless and bogus but legal SOAP requests. This is not difficult to perform if the attacker knows about the Web service interface by e.g. accessing its WSDL file or eavesdropping of a service invocation. We can imagine that the attacker can then create many mimic Web service method requests. Although someone may argue that for an attacker, it is not easy to construct a valid service request with WS-Security protection, the fact is that even the validation of the identity information embedded in a SOAP request needs XML processing time as well as additional cryptographic checks which even increases the vulnerability to DoWS attacks. According to [19], the evolution of attack sophistication in these years is towards distributed attack tools (Distributed Denial of Web Service, DDoWS), so it is not difficult to generate many SOAP requests in order to attack a specific Web service.

In the E-Business world, it is common that a series of Web services is deployed as a workflow to fulfill a complex task automatically. If these Web services are federated among several physical organizations, for example, using Liberty identity federation protocols [7, 8] or WS-Federation [9], then a DoWS attack to any of these Web services is much more dangerous than the DoS of a traditional Web site, because DoS of a traditional Web site will only influence one physical organization, whereas DoWS here will influence many physical organizations.

One obvious approach against DoWS is access control of the WSDL file of a Web service, for example, only the authorized users can retrieve this WSDL file. Unfortunately, this is not the final solution. The attackers still have many ways to get interface description of a Web service, like obtaining the information from a third party, sniffing the information on the wire to find the address and related operations of a Web service in case the Web service is not protected appropriately (such as the SOAP messages are not encrypted). Furthermore, some Web services have to embrace everyone by default, like for example, the Web services provided by Google or MSN.

Henceforth, additional measures are needed in order to defend against DoWS attacks than simply hiding the Web service by not publishing the WSDL, which in some cases is even not possible at all.

Such an additional measure to protect against DoWS attacks is presented in the following. Basically it is composed by a framework that allocates the processing capability of Web services according to the user information bound to its WSDL file, like the session information in other Http applications. With our proposed approach, even an attacker can get few valid WSDL files using the above tricks, since all Web service method invocation requests will bind the corresponding users information or sessions with them, our framework can recognize the users whom the abnormal huge request flows are from, and shut down any further connections from them. For the random requests in which the user information is invalid, our framework will shut down the connections immediately.

The rest of this paper is organized as follows. Section II summarizes the related work according to the deployment locations and indicates how it relates to DoWS. Section III presents the design of the proposed defense framework against DoWS and its components. Section IV describes an implementation of the framework in the Java/Axis based Web services hosting environment and compares the resistance capability to DoWS attacks before and after the deployment of our framework. Section V discusses some security issues related to our work and Section VI concludes.

II. Related Work

Denial of Service (DoS) and the related detection and defences have been an active area of research and development for a long time. There is much existing work focusing on DoS (however, there is little work available so far, focusing on DoWS in particular). While many of the approaches proposed so far can be reused in Web services environments, we expect to expose new approaches to attack Web Services using specific features of or semantics in Web services. In the following, we will discuss the related work by their deployment locations [10] and indicate how they relate to DoWS.

Most defense mechanisms against DoS attacks are located in the so-called victim network, i.e. the attacked/targeted network. Our work belongs to this category as well. The technologies to either detect or prevent DoS can be categorized in three main classes: signature based detection [14], abnormal pattern based detection and filtering [13], and security protocols against DoS [12, 15]. Our work is a combination of the last two approaches. We will filter connections according to user information that is recorded in the WSDL file during the retrieving stage and tagged in all request messages.

The opposite approach focuses on the source network. Here, the detection mechanisms are deployed at the source network in order to prevent network customers from generating (with or without intention) DoS attacks. For example, [11] presents a framework which is located at the access router of a network that observes the incoming and outgoing flows for each peer (the machine in this network and the machine outside this network). First, they get one normal model for each type of flow such as TCP, ICMP or UDP from the actual flow statistics, and then compare the flow pattern of each peer with the corresponding normal traffic model. The abnormal flow pattern will be identified as an attack. DoWS defenses, as an instance of [11] on the TCP layer, can benefit from this work directly, or extend it by adding a new SOAP flow type. Since our work is deployed at the victim network, we will not discuss this category further.

Intermediate network: the deployment at intermediate network can detect and prevent DoS effectively by coordination of several routers or overlay network proxies. For example, in [18], the authors use the routers to trace back the legality of every source IP before forwarding it to the destination. In [16, 17], they build a secure Internet server on an overlay network, which to some extent, avoid adding a new detection component into the routers directly that is usually unrealistic for wide deployment. Our work is not directly related to this category as well, so this category will not be discussed further.

III. Design

When accessing Web pages, there is not much work at the requestor's side except for typing some valid URLs into the Web browser. This is not true when accessing a Web service. The requestor must get a valid WSDL file for a Web service before accessing it. This two-phase based access gives us more chances to control the access of a Web service than a Web site. The underlying idea is very simple: during the retrieving stage of a WSDL file, we can customize different versions for different users by binding the user information into the WSDL file, like establishing WSDL level sessions for the users. During the accessing stage of a Web service, we can verify and filter the service requests according to the session information tagged in these requests. Thus, even when the attacker conducts a distributed denial of service (DDoS) attack, with the proposed scheme, the attack can easily be found and filtered out, since the requests belonging to the DDoS attack are from few session IDs only.

A. Architecture

Our architecture is based on the two-phase access model as shown in Fig. 1 and Fig. 2. During the first phase, when a user logs into a Web service container that hosts the different Web services and tries to retrieve a WSDL file for a specific Web service, the container will bind the related user information to a customized WSDL version for this user and return this file. A question arises here: what kind of user information should be bounded to the WSDL file and how? The most direct answer is to bind the user login information (e.g. user name) into the Web service port address location part in WSDL. This is one of the approaches to implement a session for Web applications as well.

During the second phase, when the user sends a Web service request to the target service with the session

information in the target address, the Web services container can recognize the user name by reading only the address information out from the request without invocation of any XML related processing. Based on the current work load, the filter component in Fig. 2 will decide whether or not this request can be passed into the Web services engine. The detailed algorithm is described in Section III.C. Only if the answer is yes, the request will be passed into the Web services container.

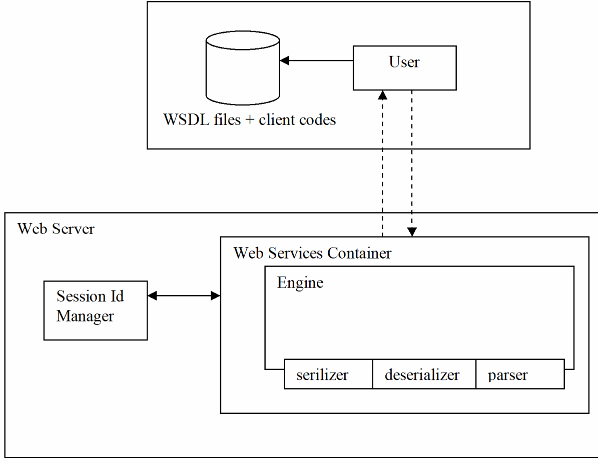

Figure 1. Phase 1: retrieving WSDL files

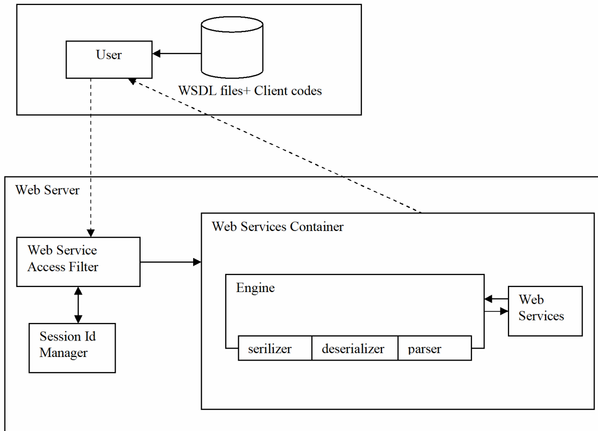

Figure 2. Phase 2: accessing a Web service

B. Session Id Manager (SIM)

SIM is used to combine the user name with the Web service port name and to hash them into a unique session ID during phase 1; and decode the incoming session ID into the original service port name in phase 2. The session IDs are used by the filter to distinguish different legal requests in addition. For example, a Web service port address of http://localhost:8080/axis/services/Version will be translated to a session id of http://localhost:8080/axis/services/Version?E6aozsMhZgAwWCHkMF1ldA==, when a user name (say Bob) is bounded to it using some hash algorithm.

C. Web Services Access Filter

A fair shared Web services system with the resistance to DoWS is our first objective. In other words, when the work load of a Web services system is beyond a threshold due to DoWS, we still need to guarantee the requests from normal requestors to come in and to be processed within reasonable time.

Now we describe the filtering algorithm towards the above objective. We extract the following parameters as the input for the filtering algorithm.

Number of total Maximum Processed Requests per unit time (NMPR): this parameter defines the maximum processing capability of a Web services system. Obvious, it depends on the underlying hardware and software, also the unit time selected. It has a constant value.

Current total Number of Processed Requests (CNPR): this parameter is obvious by its name. All of the parameter below including this one is assumed to be the value in the current unit time, so in different unit time, the values of these parameters will vary.

Current Work Load (CWL): this parameter is equal to CNMPR/NMPR.

Current Number of Sessions (CNS): this parameter will be used by the filter to decide whether or not the system is fairly used by many users or only occupied by few users in a high working load condition. The latter may indicates a potential DoWS attack.

Current Number of Processed Requests from a Session id (CNPRS): this parameter is obvious by its name.

Current Number of Maximum Permitted Requests for a user (CNMPR): the value of this parameter is dependent on other parameters, such as *CNS, CWL*, and *Threshold*.

Threshold: a value beyond which the algorithm will be switched to the DoWS defense stage.

Number of Maximum Processed Requests per unit time for any Session id (NMPRS): this parameter limits the number of the requests from the same session to a value when the work load is beyond the *Threshold*.

Number of Sessions that shows good Fairness per unit time (NSF): this parameter defines a constant value that can show that the system is fairly and normally used by multiple users.

The algorithm itself is below:

```
FilterUserReq(incomingRequest)
{// for the current unit time
  if ((incomingRequest.sessionId is in Blacklist) ||
     !(incomingRequest.sessionId is in Whitelist))
  { block this request; return;}
  if ((CWL < 1) &&
     ((CNS > NSF) ||
      (incomingRequest.sessionId is not found in the processed requests container)
     ))
  {
     Let the request come in and update the related parameters; return;
  }
  else if ((CWL < 1) &&
          (incomingRequest.sessionId is found in the processed requests container))
  {
     CNMPR = NMPR / CNS;

     if ((CWL > Threshold)  CNMPR = min(NMPRS, CNMPR);

     if (CNPRS(incomingRequest.sessionId) < CNMPR)
     {
        Let the request come in and update the related parameters; return;
     }
     else {block this request; put incomingRequest.sessionId into Blacklist; return;}
  }
  else {block this request; return;} //server overloaded
}
```

This algorithm uses three identity containers: a *Whitelist*, a *Blacklist*, and the *processed requests container*. The Whitelist stores the valid session IDs, and the Blacklist stores the compromised session IDs. The processed requests container records the number of processed requests for each valid session ID in the current unit time. It will be refreshed to be empty when the next unit time starts. The algorithm distinguishes two stages divided by a *Threshold*. When the current work load is below the defined *Threshold*, we can assume that the Web services system is still running under a low or middle level of work load, so there is little control on the incoming requests. All existing sessions will share the work load. When the work load is above the *Threshold*, our algorithm will begin to actively defend against possible DoWS attacks by monitoring and limiting the number of requests from each session to min(*NMPRS, CNMPR)*. For instance, when an attacker sends a huge number of requests to a Web service system, the filter will let them come in. When the work load is beyond *Threshold*, the filter will deny any requests from the attacker according to the session IDs within the requests. Thus, the left work load in this unit time can only be used by other requests with normal session IDs. For a concrete example, please refer to Section IV.B.

IV. Implementation And Evaluation

We have implemented this framework and deployed it in a Web services container built with Apache Tomcat [20] and Apache Axis [21]. An evaluation of our framework – performed on this implementation – shows the different resistance capabilities without and with DoWS awareness.

A. *Implementation*

First, our implementation and discussion in this section is based on the HTTP protocol that is the most common binding protocol for Web services currently. The Web services container composed of Apache Tomcat and Apache Axis uses Servlets [22] as its base. A servlet is a Java programming language class that can receive a request, process it, and return a response. In terms of HTTP, an HttpServlet can receive an HTTP request and bounce an HTTP response. An HttpServlet, by default, supports some common request methods, such as *Get*, *Post* by its doGet() and doPost() methods. For some more complicated applications, a Servlet can use the pipelined filters for both requests and responses, so that a request or a response can be customized by these filters before arriving or after leaving the servlet. Obviously, the DoWS filter we introduced in Section III.C can be deployed as a filter here. An HttpServlet can support sessions by cookies or customized URLs. Fig. 3 shows how an HttpServlet works.

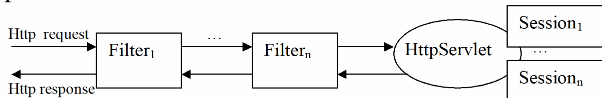

Figure 3. Basic concepts behind an HttpServlet

Apache Tomcat is a Web container programmed in Java that supports Servlets. Apache Axis is a Web services container that can process Web services requests. It has a Web service engine that can be wrapped by an AxisServlet when using Axis in Tomcat. The engine is the core of Axis, which will involve the time-consuming XML processing stages as shown in Fig. 2.

We implemented the procedure of phase 1 as proposed in Section III.A using the modified AxisServlet. In the first step, the user Bob logs-into the Web site and browses through the available Web services. In the second step, Bob selects an interesting service and downloads its WSDL definition. Finally, Bob generates the client-side codes according to the obtained WSDL file. According to the introduction in Section III, the Web service port name has been customized into a session ID for this user name.

Fig. 4 shows the components we developed and integrated into the underlying platform for phase 2. Whenever a Web service request comes in, the RequestFilter will intercept it and pass it to the Balancer. The Balancer will use the filter algorithm to make a decision. If the request with the associated session ID is permitted to come in, the RequestFilter will invoke the RequestMapWrapper that will invoke the SessionManager to translate the session ID to the original service name and replace the session ID of the incoming HTTP request with this original name. Now, the Web services engine in Axis can recognize this request and process it. If the request is refused by the Balancer, the filter will drop it directly. In the latter case, the steps 3, 4, 5 and 6 shown in Fig. 4 will not be executed.

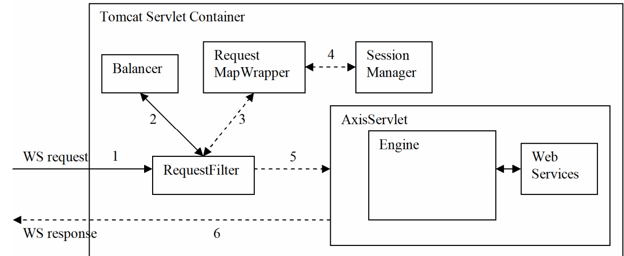

Figure 4. Components of DoWS aware Web services container

B. *Experiment Description and Result Evaluation*

The experiment is performed based on the implementation described in Section IV.A. The basic software configuration of the test-bed includes Java 1.5, Apache Tomcat 5.5.16, and Apache Axis 1.3. The test-bed consists of a set of 4 client nodes and one server node all part of a 32-node cluster. Each node is a dual AMD Athlon CPU 2.1GHz with 4GB of RAM. All of these machines are interconnected by Gigabit Ethernet and are located on the same physical switch. The 4 client nodes include 3 attack clients and a normal client each. The target Web service is the simple Version service included in Axis 1.3 as example. The service processing time for a single request is about 7 to 10 ms.

In our experiment, we allocate the values to the constant parameters of the filter algorithm as *NMPR* = 3000, *Threshold* = *0.8, NMPRS* = 120, *NSF* = 200, unit time = 1 minute. The experiment is designed so that the 3 attack clients (with different user names) flood their requests to the Web service as quick as they can. The normal client will send 120 requests to the service in about 1 minute with a constant frequency. This experiment is performed on the original Axis 1.3 without DoWS protection and on our implementation framework with DoWS protection individually. Fig. 5 shows the results.

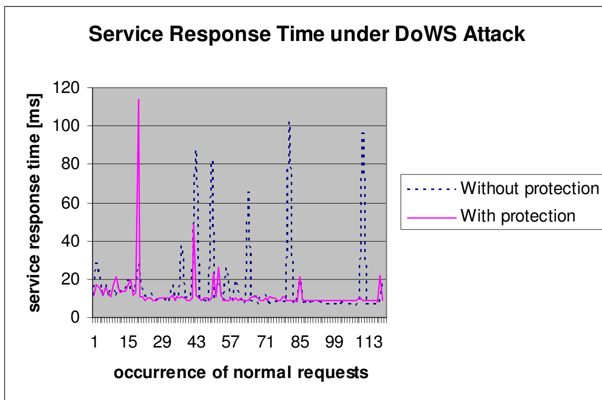

Figure 5. DoWS experiment result

In Fig. 5 the unstable response time for the normal client requests indicates that the original Axis suffers from DoWS. With our framework, the influence of DoWS is limited to a short period (for the first 20 normal requests in Fig. 5). After that, the filtering algorithm will identify the user sessions that launch this DoWS and deny their requests directly. We explain briefly how our filtering algorithm works in this experiment. First, the requests from the attackers will be processed as normal requests. The processing capability of the Version service is shared among these 4 clients. In a few seconds, the number of flooding attacking requests will get their upper bound that is 750 per session according to the algorithm. In some other cases, see 5 attacking sessions plus 1 normal session, the flooding attack requests mixed with few normal requests will occupy 80 percent (the threshold) workload of this service. At this point, the filtering algorithm will check the number of sessions and detect most requests are from few users (a sign of DoWS), and then switch to the protection stage in which the maximum number of requests from each session is limited to a reasonable value (in this experiment, it is set to 120) during the left time of this unit time. In either case above, any further requests from attacking sessions are discarded directly and only the normal requests can pass in. Thus, they still achieve good response times as shown in Fig. 5. As the last thing, the overhead introduced by our framework is very low (about 1 ms according to Fig. 5).

V. Discussion

An important building block of our defense framework is the session based filter. Our assumption is that attackers cannot sniff many valid session identities easily. The only approach is to sniff at the target Web service's side or nearby area. This is more difficult than randomly comprise some machines on the Internet and install the attacking codes as most denial of services tools do.

In the current implementation, we do not focus on who can login and retrieve a WSDL file. This is important as well because if some attackers can easily register and get many user names, they can retrieve many valid session IDs by themselves directly. A simple solution is to set up a PKI and link each user name with a certificate that can be traced back to a trusted person or organization.

Furthermore, the defense framework is not necessarily relying on retrieving a WSDL. The essential thing is to establish a session ID that can be understandable by both consumer and the target service. Alternatives to retrieving a WSDL description would be for example to put the session ID into the Service Level Agreement (SLA) or to send it by a secure email.

Finally, we need to point out that our filtering algorithm will throw a normal user into the *Blacklist* as well if the user sends the requests to a service with a high frequency in a certain unit time. During the actual deployment of our filter, some fine adjustments may be useful in order to solve this problem, such as only when one valid user is filtered out for certain times, say 3, we will treat this user as an attacker.

VI. Conclusion

In this paper, we present a framework to protect Web services against DoWS. We use a session based approach in this framework so that each valid Web service request have to present its session identity with it. On the services side, the container can recognize a user without invocation of any XML related processing step. Furthermore, we designed a fair share based filtering algorithm that can allocate the processing capability of a service among different users using the recorded session information. This filtering algorithm can guarantee that when the working load is high, the requests from DoWS will be discarded, so the normal request can still come in and get processed with a reasonable response time.

The preliminary experiment shows that our framework can effectively improve the resistance capability of a Web service to DoWS. The experiment shows as well that the overload introduced by our framework is low.

Acknowledgment

The author is thankful to Luigi Lo Iacono, Jochen Fingberg, Gregory A. Kohring, and Guy Lonsdale from the NEC C&C Research Laboratories for their valuable comments to improve this paper. This work is supported in part by the

NextGRID project (contract number 511563), which is funded by the European Commission's IST 6th Framework Programme.

REFERENCES

[1] Web Services Architecture. W3C Working Group Note, http://www.w3.org/TR/ws-arch/, 2004

[2] Web Services Description Language (WSDL) 1.1. W3C Working Group Note, http://www.w3.org/TR/wsdl, 2001

[3] WS-Security Standard V1.0, OASIS Web Services Security TC, http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf, 2004

[4] WS-Trust V1.0 Working Draft, OASIS Web Services Secure Exchange TC, http://www.oasis-open.org/committees/download.php/16138/oasis-wssx-ws-trust-1.0.pdf, 2006

[5] WS-SecureConversation V1.0 Working Draft, OASIS Web Services Secure Exchange TC, http://www.oasis-open.org/committees/download.php/16140/oasis-wssx-ws-secureconversation-1.0.pdf, 2006

[6] WS-SecurePolicy V1.2 Working Draft, OASIS Web Services Secure Exchange TC, http://www.oasis-open.org/committees/download.php/17050/ws-securitypolicy-1.2-spec-ed-01-r04.doc, 2006

[7] Liberty ID-FF Architecture Overview, Liberty Alliance Project ID-FF 1.2 Final, http://www.projectliberty.org/specs/draft-liberty-idff-arch-overview-1.2-errata-v1.0.pdf

[8] Liberty ID-WSF 1.1 specifications. http://www.projectliberty.org/resources/specifications.php#box2a

[9] Web Services Federation Language, July 2003

[10] J. Mirkovic and P. Reiher. A Taxonomy of DdoS Attack and DdoS Defense Mechanisms. ACM SIGCOMM Computer Communication Review, Volume 34 , Issue 2, 2004, ACM Press

[11] J. Mirkovic and G. Prier, and P. Reiher. Attacking DDoS at the Source. Proceedings of the 10th IEEE International Conference on Network Protocols, Washington, DC, 2002

[12] T. Aura, P. Nikander, and J. Leiwo. DOS-Resistant Authentication with Client Puzzles. Lecture Notes in Computer Science, 2133, 2001

[13] O. Spatcheck and L. Peterson. Defending against denial-of-service requests in Scout, Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation, 1999

[14] Snort. http://www.snort.org

[15] J. Leiwo, P. Nikander, and T. Aura. Towards network denial of service resistant protocols. In Proceedings of the 15th International Information Security Conference (IFIP/SEC 2000), 2000

[16] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In Proceedings of SIGCOMM 2002, 2002

[17] J. Wang, X. Liu, and A. A. Chien. Empirical Study of Tolerating Denial-of-Service Attacks with a Proxy Network. 14th USENIX Security Symposium, Baltimore, MD, July 31-August 5, 2005

[18] Y. Chen, Y. Kwok, and K. Hwang. MAFIC**:** Adaptive Packet Dropping for Cutting Malicious Flows to Push Back DDoS Attacks. *Proceedings of the 2nd International Workshop on Security in Distributed Computing Systems* (SDCS'05), Columbus, OH, June 6-10, 2005.

[19] J. McHugh. Intrusion and intrusion detection. International Journal of Information Security, July 2001

[20] Apache Tomcat Project. http://tomcat.apache.org/

[21] Apache Axis Project. http://ws.apache.org/axis/

[22] Java Servlet Technology. http://java.sun.com/products/servlet/