RC25296 (WAT1207-048) July 23, 2012
Computer Science

IBM Research Report

Web Application Server Firewall and Interactive Virtual Patching

Vugranam Sreedhar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Peng Ji
399 Keyuan Road
Zhangjiang Chuangxin
No. 10 Building, Zhangjiang High Tech Park
Shanghai 31 201203
P.R. China

Lin Luo, Shun Yang, Yu Zhang
IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100193
P.R.China

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Web Application Server Firewall and Interactive Virtual Patching

Vugranam Sreedhar
IBM TJ Watson Research Center, Yorktown Heights, NY 10598
Peng Ji
IBM China Research Lab, Shanghai, China, SH 201203
Lin Luo
Shun Yang
Yu Zhang
IBM China Research Lab, Beijing, China, BJ 100094

July 23, 2012

Abstract

A Web Application Server Firewall (WASF) is a firewall that is deployed inside a Web Application Server, such as Tomcat or WebSphere application server, to filter suspicious and malicious requests and thereby protecting the applications running on the server. In this article we focus on two key features of a WASF, called the WASP: (1) Hierarchical Rule Development (HRD) and (2) Interaction Virtual Patching (IVP). We have developed a novel fine-grained hierarchical rule schema for protecting Web applications. The hierarchical rule schema contains a number of features, including support for URI templates and RESTful requests. To improve the usability of developing and deploying firewall rules we have implemented a Rule Development Tool (RDT) that provides several capabilities for developing rules, searching for similar rules, analyzing conflicts among rules, and transforming rules from one format to another one. The RDT also provides capabilities for application developers and security administrators for interactive virtual patching of vulnerable applications. RDT allows one to semi-automatically generate rules based on importing application context for RESTful requests and searching for rules that provide protection against similar vulnerabilities.

1 Introduction

A Web application is a software application that is accessed over the Internet or an Intranet using HyperText Transfer Protocol (HTTP). In a typical Web application a client, such as a browser, interacts with a Web server by exchanging a series of messages that are made up of HTTP requests and responses. An attacker often exploits vulnerabilities that exist in a Web application to launch attacks. IBM X-Force reports[1] and OWASP[2] (Open Web Application Security Project) reports have identified some of the predominant types of attacks against Web applications that include Cross-Site Scripting (XSS), SQL Injection (SQL-I), and Cross-Site Request Forgery (CSRF) attacks. A Web Application Server Firewall (WASF) is a firewall that is deployed inside a Web Application Server, such as Tomcat or WebSphere Application Server, to filter suspicious and malicious requests and thereby protecting the applications running on the server. We have developed a new WASF called WASP (Web Application Security Protector) with advanced capabilities. WASP has several features and capabilities that goes beyond traditional firewalls. WASP provides a usable and flexible Rule Development Tool (RDT) that will allow application developers and security administrators to seamlessly and quickly develop firewall rules to virtually patch application vulnerabilities without modifying the applications source code. Rather than describing all the features and capabilities of WASP we will focus on two important aspects of WASP: (1) Hierarchical Rule Development and (2) Interactive Virtual Patching.

[1] `http://www-935.ibm.com/services/us/iss/xforce/trendreports/`
[2] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

1.1 Hierarchical Rule Development

There are two aspects to Hierarchical Rule Development. First is the Hierarchical Rule Schema (HRS) for writing flexible, fine-grained, and hierarchical firewall rules, and second is the Rule Development Tool (RDT) to quickly develop and deploy rules to protect against attacks. With the advent of modern interactive Web applications that includes AJAX and REST (Representation State Transfer) frameworks, it is imperative that a WASF is able to provide a robust defense mechanism for protecting these modern applications, where a large collection of hierarchical resources are maintained. A flexible and a hierarchical rule schema is extremely important for writing fine-grained hierarchical rules that can handle URI (Uniform Resource Identifier) templates and RESTful requests.[3] To highlight the need for a hierarchical rule schema for handling RESTful requests consider the following example:

```
http://saas.com/myPizzaShop/pizza/1/topping/3
http://saas.com/myPizzaShop/pizza/3/topping/4
```

A REST-based Web application that processes the GET requests for the above URLs will typically not create one static page for each resource. The Web application will construct a layout for the URLs using URI template. A URI template is a mechanism that allows one to specify a URL to include parameters that are substituted before the URL is resolved. Using URI templates an application can create the following the template for the above example: `http://saas.com/myPizzaShop/pizza /{pid}/topping/{tid}`, where `{pid}` and `{tid}` are resource variables that map to `1` and `3`, respectively for the first URL and map to `3` and `4`, respectively for the second URL. In order to develop rules for each resource one requires ability to express fine-grained and hierarchical rules. In other words, to handle such URI template-based HTTP requests requires an ability to model hierarchical rules and also an understanding of back-end application structure. Using our hierarchical rule language, we model the above URI template as follows: `http://saas.com` represents the instance of `Hostname`, `pizza`, `{pid}`, `topping`, and `{tid}` are instances of `FilePathNode`. For resources `{pid}` and `{uid}` we also set the attribute `isVariable` to be `true` and the `variableExpression` is set `ˆ[0-9]+$` indicating that it matches numerical value pattern. Another point is that we can set the attribute `inheritParent` to the node `tid`, which means the rule bound to the parent node such as `topping` will be inherited and applied to children nodes. The above URI template is represented using JSON (JavaScript Object Notation). JSON is a simple and flexible language that we use for representing not only core elements of our rule language, but also to express meta-information about various parts of the rule elements to provide additional capabilities.

One difficulty that most security administrators face is how to go about developing firewall rules. We have built a Rule Development Tool (RDT) that allows security administrator to quickly develop rules. The RDT also provides many other features related to the development of firewall rules, including importing application context, rule transformation, rule testing and deployment, and rule analysis.

1.2 Interactive Virtual Patching

Imagine you are an application developer or a security administrator and you just deployed a new version of an application on a Web server. Let us assume that you run an application vulnerability scanning tool, such as IBM Rational AppScan Tool, and found a vulnerability in the new version of the application. Unfortunately due to software application release cycle you may not able to patch the application and redeploy it. Virtual patching is a process in which a security administrator or an application developer will develop and deploy one or more rules on a WASF to prevent any exploitation of application vulnerability in a released Web application product. Virtual patching is not a substitute for secure coding, but it augments secure software process to prevent exploits when software has to be rushed to release or has been released. In Software as a Service (SaaS) and Cloud environments where multiple applications are hosted, virtual patching will become extremely important. Virtual patching will bring added security, especially for patching third party applications. In this article we describe in detail how we can semi-automatically generate and provision rules using application context and black-box testing tool results, and develop a method for interactive virtual patching (IVP) as part of development life cycle. We will collect the application context information

[3] `http://tools.ietf.org/html/draft-gregorio-uritemplate-08`

such as web.xml, struts.xml or annotation information, and then use those to generate the RESTful URI template. To illustrate an example of virtual patching we show below a snippet of AppScan scan tool message that is injected and processed by a back-end application.

```
GET /agora/meetings/util/publishFile.jsp?id=d44a9d";
</script><script>alert(62415)</script>&load=1 HTTP/1.1
```

The AppScan tool will intercept the response from the back end application and will observe that the following snippet in the response message which indicates that the input was not validated, and so the vulnerability can be exploited to launch XSS attack.

```
"d44a9d";</script><script>alert(62415)</script>"+"&fileId="
```

Using our IVP the security administrator can quickly develop a rule to prevent such input validation vulnerability and thereby prevent the XSS attack. The core value of our IVP is the tooling support that a security administrator can use to quickly develop and deploy rules to prevent exposure of application vulnerability. Using our Rule Development Tool (RDT) a security administrator can import application context for URL templates, run AppScan Tool, search for rules that prevent similar vulnerability, test the rule, extend and improve the rules, and then deploy the rule for real time monitoring. The whole process of scanning, rule development, and testing is interactive and can be part of an application development life cycle.

In the rest of the article we will describe in detail the aforementioned features of WASP. We have implemented WASP using J2EE framework, and so we can deploy our WASP inside any J2EE Web application server. We have integrated WASP into WebSphere application server (WAS) and Tomcat server and deployed it in a Cloud environment. We will present preliminary empirical results. Section 2 presents the WASP architecture design. Section 3 describes the Hierarchical Rule Development. Section 4 discusses the Interactive Virtual Patching. Section 5 presents preliminary experiment results. Section 6 highlights related work and Section 7 concludes the article.

2 WASP Architecture

In this section we briefly highlight the different compoenents of WASP. Figure 1 shows the overall system architecture of WASP. WASP consists of four key modules: WASP Server Engine Plugin (WSEP), WASP Rule Repository and Management (WRRM), WASP Analyzer (WA) and WASP Logging Logging and Monitoring Portal (WLMP). WASP Connector component of WSEP provides functionality for integrating WASP to different Web application server such as IBM Websphere, Tomcat, Apache, Microsoft IIS and so on. WASP Engine is the core component of WSEP which contains WASP Rule Controller (WRC) to validate and filter message content. Both inbound and outbound messages are copied by the WASP Connector and passed to the WASP Engine for analysis. The WASP engine first constructs an internal hierarchical message model after normalization. WASP rules are then applied to different parts of message hierarchy model. Depending on the rule semantics the WASP Engine will determine whether to block or pass the message to either back-end application for inbound messages or outbound messages for client browsers.

WRRM is a centralized repository to manage WASP rules. For Cloud environment, the repository can be extended as a centralized shared service to manage rules and ensure that the latest rule changes can be adopted by all WASP instances. Although we have not described in this article, WRRM can provide capabilities for managing and optimizing WASP rules across multiple WASP instances and dispatch instance specific rules to a specific WASP instance running on a specific Web Application Server. WLMP provides capabilities to monitor and log exception alerts. We can export the alert logs to generate summary reports for executives. WA is used to perform analysis on not only logs of WASP instances, but also other external resources such as vulnerability scanning reports and configuration information of web application containers (e.g. web.xml, application.xml, etc.) to model RESTful URL templates.

3 Hierarchical Rule Development

In this section we describe the Hierarchical Rule Development (HRD) feature of WASP. HRD consists of two main parts. First is the Hierarchical Rule Schema (HRS) that allows rule developer and security administrator to develop

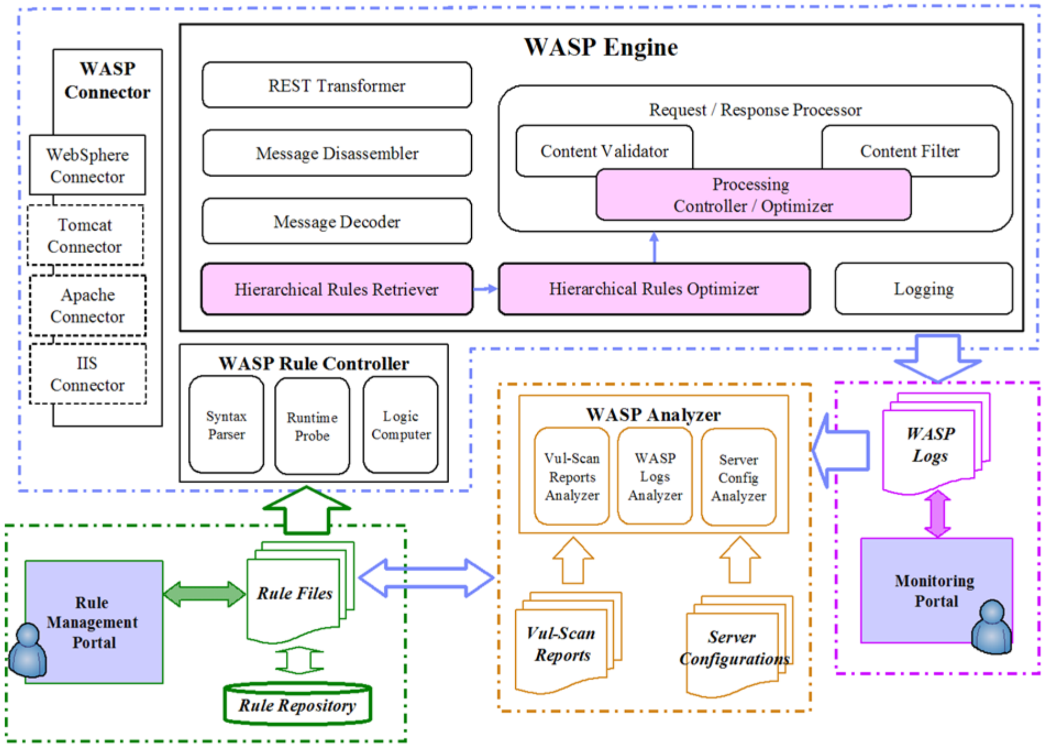

Figure 1: WASP System Architecture

fined-grained hierarchical positive and negative rules. Second is the integrated Rule Development Tool (RDT) that allows security administrator and application rule developer to perform various activities related to the development of rules such as search for similar rules, upload application configuration for dealing with RESTful requests, testing rules, checking for consistency of rules, and deploying rules.

In the rest of this section we will explain and highlight some novel features of HRS. Rather than developing a new surface syntax to express rules, in WASP we use JSON (JavaScript Object Notation) as our underlying representation for rules.[4] The HRS consists of three main parts: (1) HTTP Message Model that define the core model of HTTP message structure, (2) Rule Model that can be used for writing firewall rules, and (3) Message Rule Binding which is necessary to determine the set of rules that should be triggered at runtime for a given message.

3.1 HTTP Message Model

The basic format of HTTP request and response messages consists of (1) initial line, (2) sequence of header lines, (3) a new blank line, and (4) body. The initial line for request message contains one of HTTP methods (such as `GET`, `POST`, `HEAD`, etc.). The initial line for response message contains status information that includes status code. Figure 2 illustrates an example of a POST request. The overall HTTP message model using UML (Unified Modeling Language) is shown in Figure 3. The root of the HTTP message model is the abstract element called `Message`. The `RequestMessage` and `ResponseMessage` are concrete types of `Message` element, and they correspond to HTTP request and HTTP response messages. Recall that the initial line of an HTTP request contains method, URL, and version; and we represent them as `Method`, `URLTemplate` and `Version` elements. The `Version` element is

[4] We also support XML schema representation, but we will not discuss the XML schema representation in this article.

```
POST http://chinabank.com:8080/Account/transferMoney.php
    HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg
Referer: http://9.186.54.51:8080/MyBankApp/request.jsp
Accept-Language: zh-cn,en-us;q=0.5
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
   Windows NT 5.1; SV1; .NET CLR 2.0.50727;
   .NET CLR 3.0.04506.648; .NET CLR 3.5.21022; CIBA)
Host: 9.186.54.51:8080
Content-length: 19
Pragma: no-cache
Cookie: JSESSIONID=29D07EE24B69CC4C4173F9AFBF87A6A

acct=BOB&amount=100
```

Figure 2: Example of an HTTP Request

used to represent HTTP version number in both HTTP request and response and so it is part of in `Message` element (denoted using solid diamond symbol). A Method element can be one of GET, POST, PUT, DELETE, and HEAD.

Each HTTP request and response contains a header section, and we model this using the `Header` element. We consider the `Header` element to be a set of name-value pair represented as a set of `Parameter` elements. Notice that in our model we explicitly model cookie using the `Cookie` element; and the reason for this is that they contain important elements (such as session information and authentication information) that are needed for writing filtering rules against cookies. Once again a `Cookie` element contains a set of name-value pairs and so we model them as a set of `Parameter` elements. The `QueryString` element, which is part of `URLTemplate` element, is also modeled as a set of name-value `Parameter` elements. Finally when the value of the `Content-Type`, defined in the HTTP request header, is `application/x-www-form-urlencoded`, the message body will also be name-value pairs, and therefore we model them as a set of `Parameter` element. The `Content-Type` element defines the content type of the body. Unlike `RequestMessage` element, `ResponseMessage` element contains a `ResponseStatus` element. The `statusCode` attribute can be any of the status code as defined by IETF HTTP protocol standard. The `ResponseMessage` element also contains `Header` element, `Version` element, and `Body` element.

3.2 Hierarchical URL Model

Next let us focus on the URL elements of HTTP request, which we model it as a `URLTemplate` element. As shown in Figure 3, each `URLTemplate` element contains three parts: (1) `Hostname` element, (2) sequence of `FilePathNode` elements, and (3) `QueryString` element. The `QueryString` contains `Parameter` element that represents the name-value pairs. Now consider the following HTTP request URL `http://chinabank.com:8080/Account/transferMoney.php?acct=BOB&amount=100`. In this example `http://chinabank.com:8080/` is an instance of `Hostname` element. The resources `Account` and `transferMoney.php` are instances of `FilePathNode` element. Finally `acct=BOB&amount=100` is an instance of `QueryString`. Notice that `QueryString` contains two name-value pair of Parameter elements separated by `&:` `acct=BOB` and `amount=100`. Each `FilePathNode` contains several attributes such as `isVariable`, `variableExpression`, and `inheritParent`, and we will explain them in later sections. To summarize, our HTTP message model is concise and is semantically rich for developing rule model. The example in Figure 4 illustrates JSON representation of HTTP message model instance for the example shown in Figure 2.

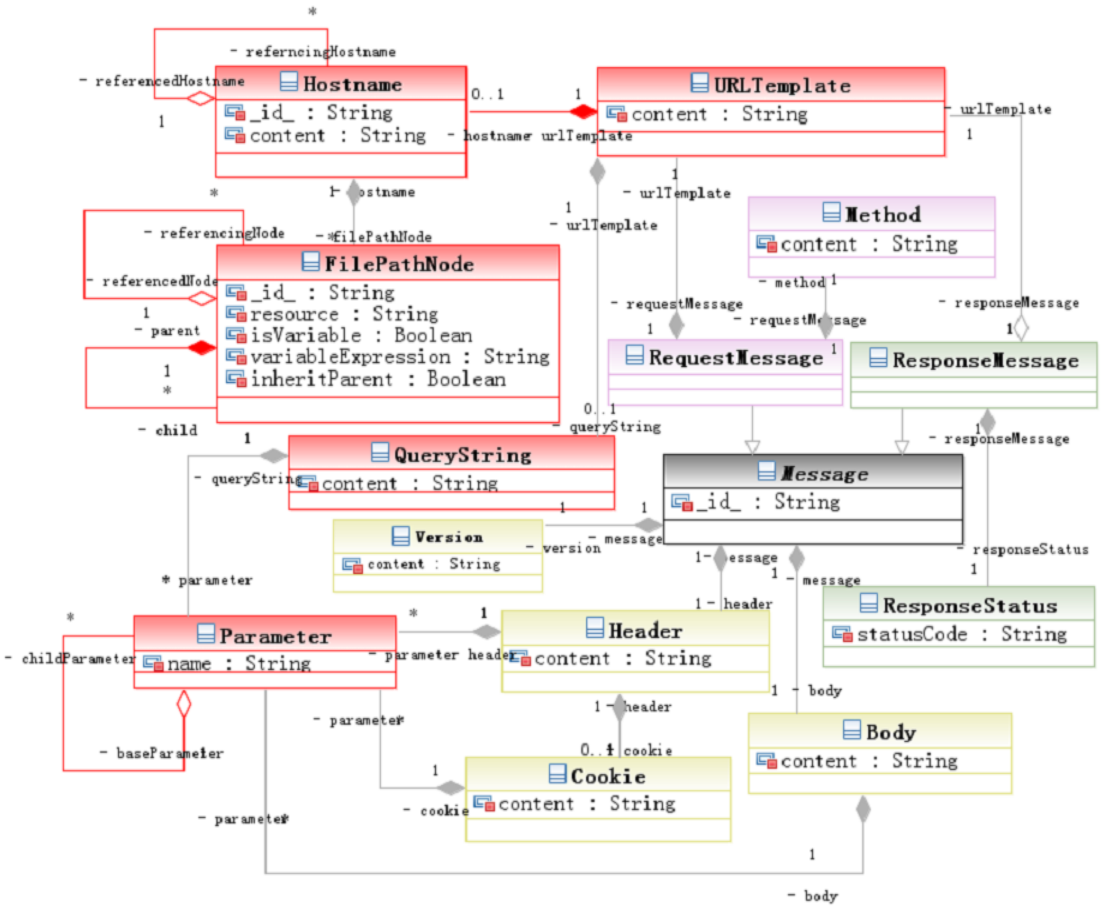

Figure 3: HTTP Message Model

3.3 Rule Model

Figure 5 shows the UML class diagram for the HRS rule model. A security administrator can use RDT for developing rules. Figure 9 shows the screen-shot of rule development using RDT. There are three main parts in the rule model. The first part is made of `Rule` and `RuleSet` elements, the second part is made of `Condition` element, and the last part is the `Action` element.

3.3.1 Rule and RuleSet element

The basic structure of a rule will look like `R01: if condition then action`. Whenever the condition holds true the corresponding action is executed. The following is an example of a rule `R01: if (NUM.GT(STR.Length( Request.Header.Parameter[Content-Length]), 100)) then Action.Log`. In the above rule `Request.Header.Parameter[Content-Length]` identifies a particular header parameter and we check if its string length is greater than 100. Notice the way we access HTTP message model elements. Recall from Figure 3 that `Parameter` is a name-value type of model element and `Content-Length` is the name and the notation `Parameter[Content-Length]` returns its value. An instance of a `Rule` element has three main parts: (1) Rule identifier that identifies the rule, such as `R01`, (2) Condition such as `NUM.GT(STR.Length (Request.Header. Parameter[Content-Length]), 100)` and

```
"requestmessage":[
{ "_id_" : "M01",
"method": {"content": "POST"},
"version": {"content": ""},
    "urltemplate":{"content":"http://chinabank.com:8080
                         /Account/transferMoney.php",
       "hostname":{"_id_": "H01",
                    "content":"chinabank.com",
                    "filepathnode":[{"_id_": "FPN01",
                         "resource":"Account",
                         "filepathnode":[{"_id_": "FPN02",
                           "resource":"transferMoney.php",
                           }] }] } },
"header": { "content": "",
          "parameter": [ {"name": "Accept"},
                         {"name": "Referer"},
                         {"name": "Accept-Language"},
                         {"name": "Content-Type"},
                         {"name": "Proxy-Connection"},
                         {"name": "User-Agent"},
                         {"name": "Host"},
                         {"name": "Content-length"},
                         {"name": "Pragma"} ],
   "cookie": {"content": "",
              "parameter": [{"name": "JSESSIONID"}], } },
"body": { "content": "",
          "parameter": [ {"name": "acct"},
                         {"name": "amount"},], } },]
```

Figure 4: JSON Representation of HTTP Request Model

(3) Action, such as `Action.Log`. Our WASP rule schema contains several meta-information, such as `name`, `_id_`, `description`, etc., which are useful for writing rules, and we use JSON to write rules. We group together a set of rules that have some common purpose using `RuleSet` elements.

3.3.2 Condition and Action element

We use `Condition` element to model rule condition. We support a number of different condition operators such as regex matching, numeric, string comparison, and external plugin extensions to express complex condition logic (see Figure 7). We use `Action` element to model actions of a rule (see Figure 8). A rule can trigger more than one action when the corresponding condition of the rule is satisfied. The attribute `actionType` is used to represent the action type. The action types such as `Block`, `Allow`, and `Log` are straightforward to understand. The `Record(variable,value)` is useful when the runtime engine want to manage state across different rule execution, for instance, supposing we want to know if a particular rule `R01` had fired previously. In this case, when rule `R01` is fired we will record its status in a variable using `Record(variable,value)` action. The action type `Execute(path)` will execute and external command referenced by a fully qualified path. The action type `Rewrite` is useful to rewrite values of certain elements, such as rewrite the URL value.

3.4 Rule Binding

Based on the HTTP message model and the rule model, the next concept that is essential for a security administrator is to understand how to bind rules to HTTP messages. Security administrator will use our RDT to bind rules to HTTP message models. Figure 10 illustrates a screen-shot of rule binding using RDT. The first step in the RDT is to create a new template based on HTTP message model. The RDT will present the new template of HTTP message model to a security administrator and the template will highlight all the elements of the HTTP message model. There are one or more entries for each element that can be filled by the security administrator. For instance, for the URLTemplate

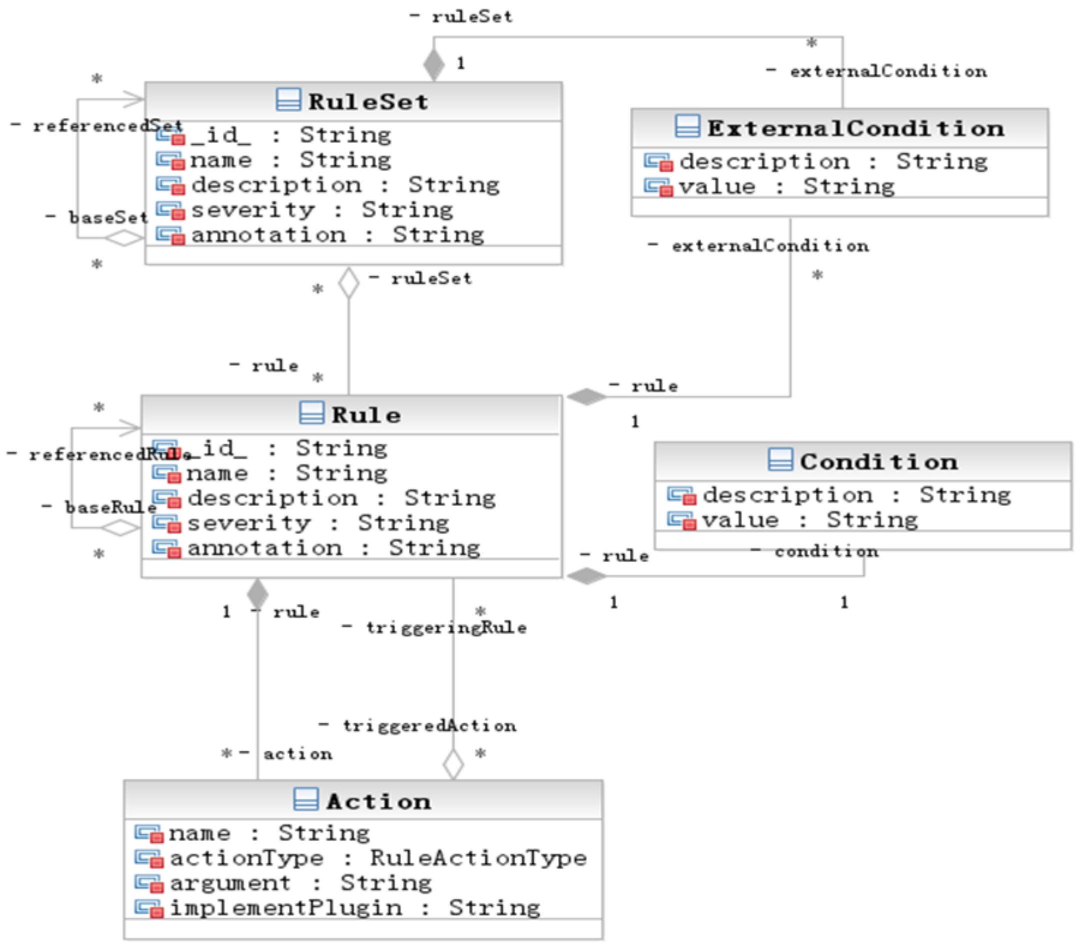

Figure 5: Rule and Rule Set Model

element, the security administrator can fill in a URL that includes host name and file path nodes. The file path node could contain resource variables such as {`pid`}, as we discussed earlier. We will use the URLTemplate instance as an index or key during runtime to retrieve the current HTTP model/template instance. For the `Header` element we can also create the set of headers that are allowable for the current HTTP model instance. Once the instances of all the elements of a new template is created the security administrator can then bind or associate one or more rules to each element by either using pre-existing set of rules or by developing new rules.

3.5 Rule Chaining

WASP supports a more expressive rule chain semantics compared to ModSecurity. Consider the following ModSecurity rule chain:

```
SecRule REQUEST_METHOD ^GET$ chain, setenv:var1=true,deny
SecRule REMOTE_HOST ^127\.0\.0\.1 chain, setenv:var2=true
SecRule REQUEST_URI /denied.html setenv:var3=true
```

In ModSecurity, the disruptive action, DENY, is restricted to the first rule in the chain. It may therefore lead to an unexpected side-effect where a request is denied only when all three rules in the chain trigger and all three non-

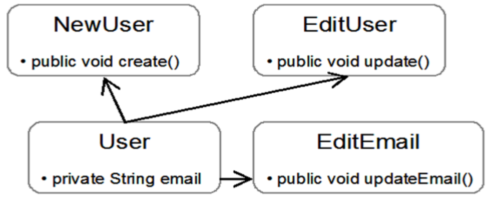

Figure 6: Dependency of Servlet classes

disruptive actions that set variables will be executed. We support a logical semantics where the DENY action is defined in the last rule of the chain. In WASP rule chain design, (1) rules are chained in a tree-like fashion using TRIGGER action; (2) using forward chaining semantics, one or more conditions that are shared within rules are considered to be chained, and the same condition is evaluated only once in the chain; (3) all actions are performed as though they are standalone rules and each of them can hold more than one TRIGGER actions. Figure 11 shows the rule chain using JSON for the above ModSecurity rule chain. Notice that only the first rule is the entry rule and the rest of the rules are chained and they will be triggered as a consequence of the chaining.

3.6 Multiple Deployment

An important aspect of WASP is that a Web application can have several different URLs for different deployments, as illustrated in Figure 12. Also certain URLs can be configured as a "base URL" on which other URLs can be depended. Most modern Web applications use configuration files (web.xml, struts.xml) to map resources to URLs. For the XML configurations illustrated in Figure 13, the application has three separate URLs `/user/register`, `/user/update` and `/user/updateemail` that map to three servlet classes `UserManagement.NewUser`, `UserManagement.EditUser` and `UserManagement.EditEmail`, respectively. The servlet classes are implemented as POJO classes as illustrated in Figure 13. The class `User` is used in both classes `NewUser` and `EditUser`, and the attribute `email` from `User` is used in the class `EditEmail`. Thus, the WASP rules for `/user/register` and `/user/update` will be the same since the mapped servlet classes both operate on the same POJO class instance. Also, the rule for checking email attribute on `/user/register` and `/user/update` will be shared too. URL references, where the reference logic can be viewed from different perspectives.

3.7 Hierarchical Model and Inheritance

Web application developers often want the ability to express the layout of URLs that their application can respond. To further elaborate, consider the following URLs that a particular Web application will handle:

```
http://saas.com/myPizzaShop/pizza/1/topping/3
http://saas.com/myPizzaShop/pizza/3/topping/4
```

The Web (REST) application that processes the GET requests for the above URLs will typically not create one static page for each resource. The Web application will construct a layout for the URLs, and using URLTemplate one can create the template as `http://saas.com/myPizzaShop/pizza/{pid}/topping/{tid}`, where `{pid}` and `{tid}` are resource variables. We model the above URLTemplate using our model as follows: `http://saas.com` represents the instance of `Hostname`, `pizza`, `{pid}`, `topping`, and `{tid}` are instances of `FilePathNode`. For resources `{pid}` and `{uid}` we also set the attribute `isVariable` to be `true` and the `variableExpression` is set `^[0-9]+$` indicating that it matches numerical value pattern. The above URLTemplate can be represented using JSON as shown in Figure 3.7.

Next we discuss how to inherit rules that are written for parent parts of the HTTP model. To understand the notion of parent we will use URL structure to build a URL tree model. The motivation for inheritance is that most Web applications have hierarchical structure. For instance, the web.xml file, which contains the configuration of a Web

Numeric	**EQUALS**	**GT**	**LT**
	GE	**LE**	
String	**EQUALS**	**LENGTH**	**CHARAT**
	CONTAINS	**STARTWITH**	**ENDWITH**
Boolean	**AND**	**OR**	**NOT**
	EQUALS		
Regular Expression	**MATCH**	**UNMATCH**	**MATCHALL**
	NONMATCHALL		
Content Validation	**HTML.MatchTag**	**HTML.MatchAttributeValue**	**HTML.MatchAttributeName**
	JSON.MatchTag	**JSON.MatchValue**	
	XML.MatchTag	**XML.MatchTagByAttributeValue**	**XML.MatchAttributeValue**
	XML.MatchAttributeName		
Plugin	**INVOKE**		

Figure 7: Condition operators in WASP

application, describes a tree-like structure for a Web application. To illustrate this we further consider the following URLs that are part of a sequence of HTTP messages:

```
http://saas.com/myPizzaShop/pizza/order.php?quatity=1&type=cheese
http:  //saas.com/myPizzaShop/pizza/enquire.php
http://saas.com/myPizzaShop/pasta/order.php?size=1&type=meat
```

A security administrator can write a set of rules for the resource `pizza` and this set of rules could apply to all descendant of `pizza` node in the above URLs. In other words, both `order.php` and `enquire.php` can inherit this rule. Now when an HTTP message with URL `http://saas.com/myPizzaShop/ pizza/enquire.php` arrives we will construct an instance of HTTP message model. The run time will then use the URL to access the message model for `enquire.php` and if the `inheritParent` attribute is set true for the file path node `enquire.php`, the rule bound to `pizza` will be inherited and applied to HTTP message model elements.

3.8 Rule Transformation

In our RDT we provide capabilities to transform firewall rules written for one WASF to our HRS. In Figure 15, we illustrate the rule sample from ModSecurity rules. For example, the `ARGS` in ModSecurity is the predefined label to denote the parameter name, and the `REQUEST_FILENAME` is used to denote the request URL. In this rule sample, it uses the `rule chain` to control the input value for the parameters `id` and `email` in the request URL as `"/agora/meetings/util/publishFile"`. By using HRS, we can easily transform the ModSecurity rules in Figure 15 to the WASP rules in Figure 3.8. Especially, in WASP rules, the friendly error message can be defined as one action in the WASP rule element. For example, the `POS_CHECKALPHANUMERIC` rule in Figure 3.8 can enable

Action Name	Action Parameters
Allow	Null (current message)
Block	Null (current message)
Log	Null (current message)
Rewrite	OldValue, NewValue
	Section, OldValue, NewValue
	Section, NewValue
Record	Variable, Value
Trigger	Rule
	RuleSet
Execute	Command Path
HTML_RemoveTag	Tag, Condition
HTML_RemoveTagByAttributeValue	Tag, Attribute, Condition
HTML_RemoveAttributeByName	Tag, Condition
HTML_RemoveAttributeByValue	Tag, Attribute, Condition
HTML_SanitizeAttribute	Tag, Attribute, Condition
XML_RemoveTag	Tag, Condition
XML_RemoveTagByAttributeValue	Tag, Attribute, Condition
XML_RemoveAttributeByName	Tag, Condition
XML_RemoveAttributeByValue	Tag, Attribute, Condition
XML_SanitizeAttribute	Tag, Attribute, Condition
JSON_RemoveTag	Tag, Condition
JSON_RemoveValue	Tag, Condition
JSON_SanitizeValue	Tag, Condition

Figure 8: Action operators in WASP

the friendly error message to WASP client when some violation is detected in WASP server.[5]

4 Interactive Virtual Patching

Virtual patching is a process in which a security administrator or an application developer will develop and deploy one or more rules on a WASF to prevent any exploitation of application vulnerability in a released Web application product. Virtual patching is not a substitute for secure coding, but it augments secure software process to prevent exploits when software has to be rushed to release or has been released. In Software as a Service (SaaS) and Cloud environments where multiple applications are hosted, virtual patching will become extremely important. Virtual patching will bring added security, especially for patching third party applications. In this section we will show how to provision rules using application context and black-box testing tool results to develop a method for interactive virtual patching (IVP) as part of development life cycle. Our IVP for provisioning rules consists of the following steps. In the rest of this section we will explain each step in detail.
Step 1: Generate templates for WASP rules using WASP rule schema and application context information.
Step 2: Generate vulnerability report using AppScan testing tool.
Step 3: Generate WASP rules using rule templates and AppScan vulnerability report.
Step 4: Test and deploy the rules to run on WASP engine.

[5]Friendly error feature of WASP is described in detail in a companion article [].

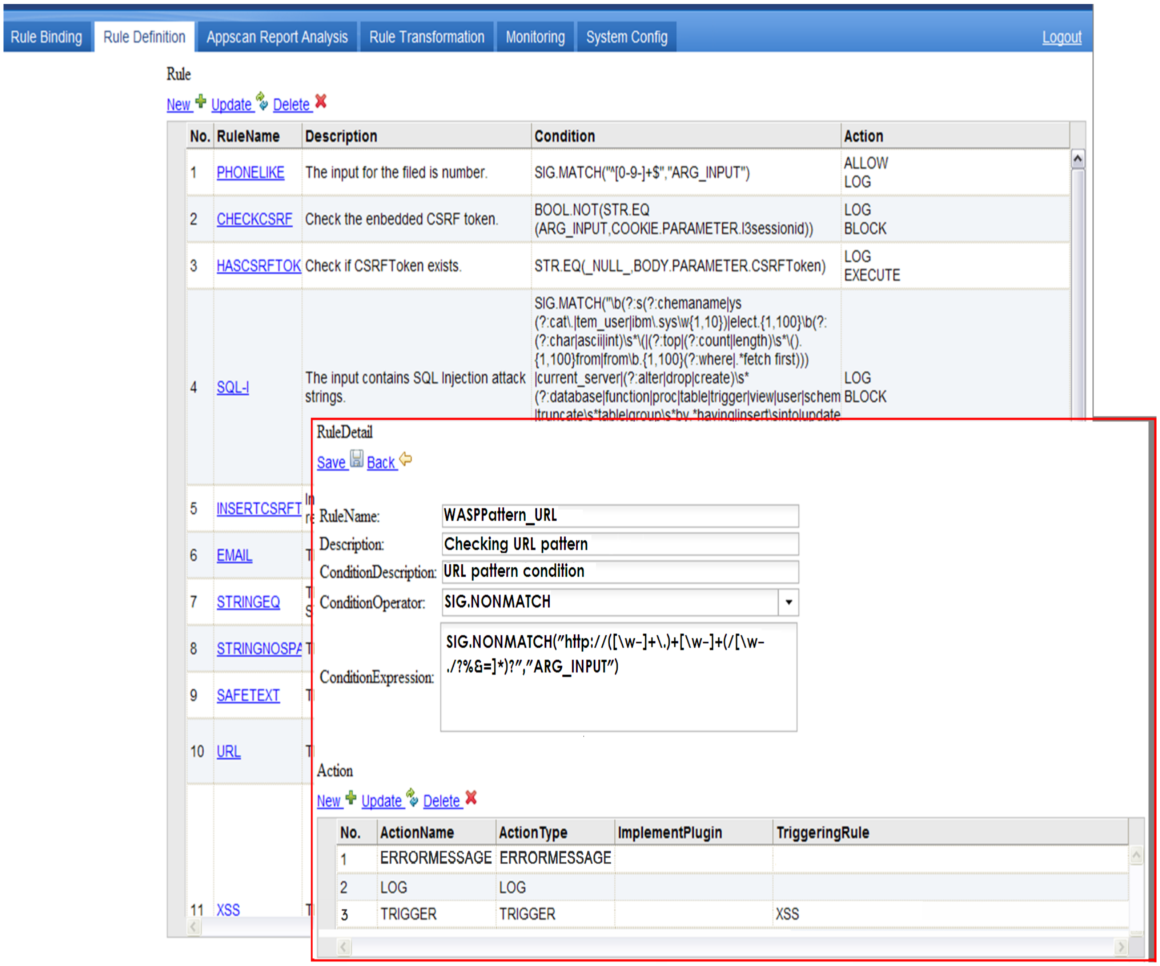

Figure 9: Rule Model in Rule Development Tool

4.1 Template Generation

The rule template generation uses two sources of information as shown in Figure 4.1. First is the WASP HRS, which provides relationship among various HTTP elements. Second is the context information such as web.xml, struts.xml or expressed as application annotations, which provide the hierarchy structure of the Web application. Web application developers often want the ability to express the layout of URLs that their application can respond. The layouts are often expressed in a configuration file, for example in Figure 4.1, the Web application will construct a URLTemplate as `/agora/{meetingId}/shares`, where `{meetingId}` is resource variables.

4.2 Black Box Testing and Analysis

IBM Rational AppScan is a robust Web application black-box security testing tool that can be used to scan applications for identifying vulnerabilities. AppScan has built in capabilities to scan and test of a wide range of Web application vulnerabilities, including those identified by OWASP and IBM ISS XForce team. We have integrated AppScan within WASP framework so that a security administrator or application developer can configure and scan Web applications directly using WASP framework. In this section we will briefly go over the key elements of AppScan vulnerability report and highlight how we analyze them for generating WASP rules. AppScan reports typically include two types

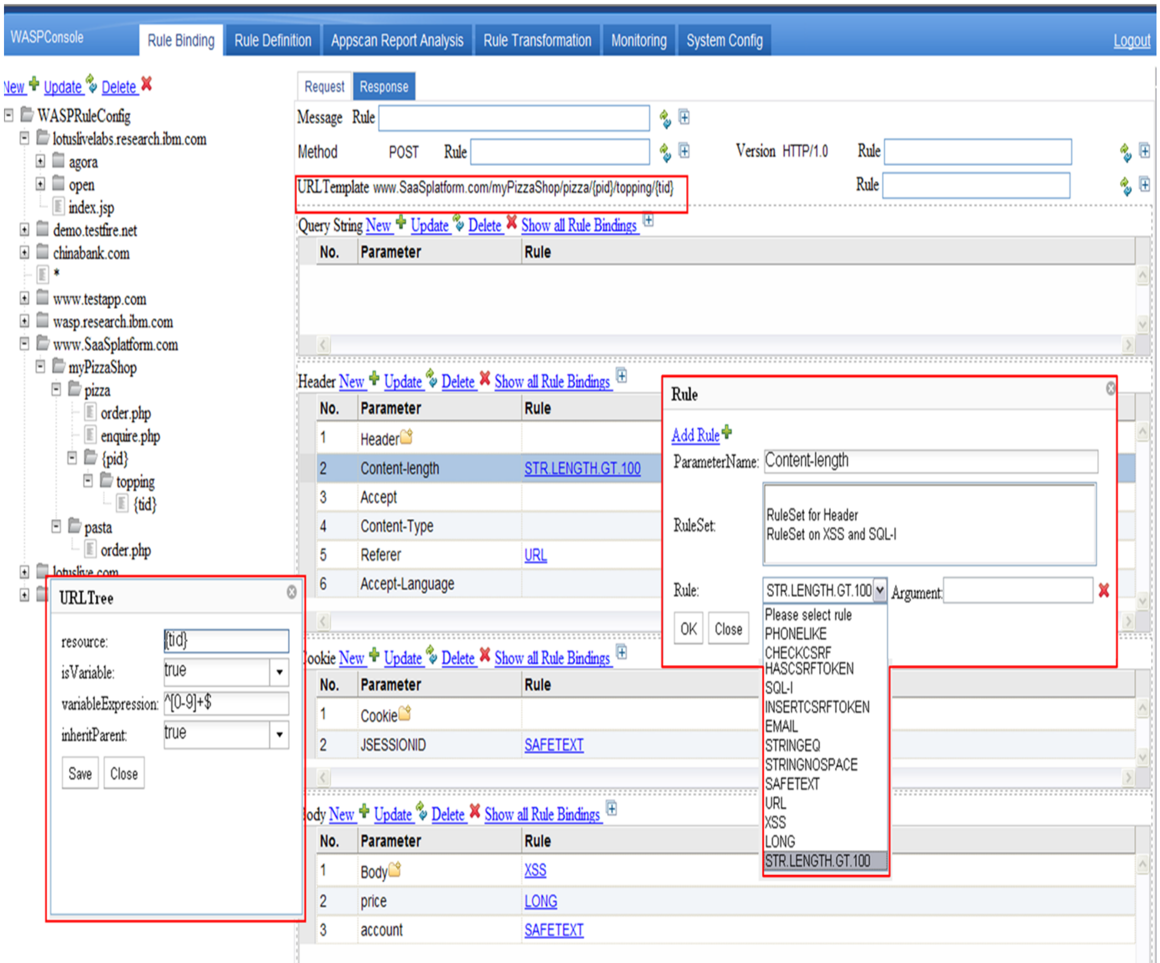

Figure 10: HTTP message model and rule binding in Rule Development Tool

of security issues:

Application security issue is mainly due to insecure code implementation. In WASP we focus mostly on application security issue.

Infrastructure security issue is mainly related to improper system configuration or system-level holes. We recommend fixing this kind of issue by changing the system configuration and updating the patches accordingly because it will not cost much time like revising application codes. When there are limitations to update system configuration we can still utilize WASP to patch the vulnerability.

To highlight the capability of WASP virtual patching let us focus on the vulnerable URL `https://lotuslivelabs.research.ibm.com/agora/meetings/util/publishFile.jsp`. Looking at the URL it seems that it provides utility to publish files. AppScan reported a possible Cross-Site Scripting (XSS) vulnerability for this URL. Consider the HTTP request message shown in Figure 4.2 that is injected by AppScan testing tool and processed by Agora backend application:

Notice the parameter `id=d44a9d";</script> <script>alert(62415) </script>`. AppScan injected this script and is later verified that the corresponding HTTP response contains the injected script. AppScan testing tool alerts such cases as being XSS vulnerable URL. We can use WASP to block such HTTP requests using virtual patch rules.

```
[{ "rule": [
  { "_id_": "R01",
    "name": "URIVar1",
    "description": "If URI ends with denied.html,
             then set variable var3 as true",
    "condition": { "value": "STR.ENDWITH(\"REQUEST.URI\",
             \"/denied.html\")" },
    "action": [
       { "actionType": "RECORD",
           "argument": ["var1", "true"] },
       { "actionType": "TRIGGER",
           "triggerRule": [ "R02" ] }
       ]},
   { "_id_": "R02",
     "name": "RemoteHostVar2",
     "description": "If remote host equals 127.0.0.1,
              then set variable var2 as true and trigger Rule R03",
     "condition": { "value": "STR.EQUALS(\"REQUEST.REMOTE_HOST\",
                  \"127.0.0.1\")" },
     "action": [
        { "actionType": "RECORD", "argument": ["var2", "true"] },
        { "actionType": "TRIGGER","triggerRule": [ "R03" ] }
        ]},
   { "_id_": "R03",
     "name": "RequestURIVar3",
     "description": "If method equals GET,
                     then set variable var1 as true",
     "condition": { "value": "STR.EQUALS(\"REQUEST.METHOD\",
                   \"GET\")" },
     "action": [
        { "actionType": "RECORD", "argument": ["var3", "true"] },
        { "actionType": "TRIGGER", "triggerRule": [ "R04" ] }
        ]},
    { "_id_": "R04",
      "name": "DenyRule",
      "description": "This is a rule to deny message.
                    It will be triggered in the rule chain.",
      "condition": { "value": "BOOL.TRUE" },
      "action": [
         { "actionType" :"DENY" }
         ]}
   ],
   "requestmessage":[
     { "_id_" : "AllMsg",
        "urltemplate":{ "content": "*" },
        "binding":[{"rule": [ "R01" ]}]
     }
    ]
 }]
```

Figure 11: An example illustrating rule chaining

Category	Requirement	Example 1	Example 2
Hostname (for same application)	Different Hostname	http://tieba.baidu.com	http://bbs.baidu.com
	Different IP	http://9.186.56.213/WASP/WebConsole	http://9.181.54.52/WASP/WebConsile
	Different protocols	http://www.mybank.com/pay.jsp	https://www.mybank.com/pay.jsp
	Different ports	http://www.mybank.com:80	http://www.mybank.com:8080
FilePathNode (for same module)	Different node name	http://www.mybank.com/deposit/action.do	http://www.mybank.com/cunqian/action.do
	Different node sequence	http://www.SaaSplatform.com/myPizzaShop/pizza/{pid}/topping/{tid}	http://www.SaaSplatform.com/myPizzaShop/topping/{tid}/pizza/{pid}

Figure 12: Samples of Multiple Deployment Configurations

4.3 Rule Generation

Recall the URLTemplate for the Rest applications can be obtained by generating the rule template in Step 1, which parses the web.xml, struts.xml etc. Second, we parse the vulnerability issues in AppScan XML report to create WASP rule instances based on WASP rule schema, and then merge the URLTemplate value to the rule instances. Because of the false positive issues, the security admin need to approve the rules generated from AppScan report as shown in Figure 4.1, and then the rules will be imported to the runtime WASP engine for further testing.

From the AppScan report we notice that the URL query parameter id is susceptible to XSS vulnerability. In WASP, we can use either negative rules or positive rules to handle XSS vulnerability. In order to explain better, we just put two rules in Figure 3.8 as example to show how to binding rules to the element.

4.4 Testing and Deployment

A security administrator can use WASP Rule Development tool to develop rules and can then test them inside the WASP framework. We can inject the HTTP message generated by AppScan in our testing environment. The WASP runtime engine will process the requests and then generate log files that contain what rules were fired and what actions were taken. The WASP monitoring console for the violation logs is shown in Figure 4.1. The security administrator can, if needed, refine the rules interactively until he or she is satisfied. Once the rules have been tested within WASP test environment it can be deployed for real time monitoring of HTTP requests and responses. The WASP test environment is still under development and more details of this will be provided later in the project milestone.

5 Empirical Evaluation

We performed two kinds of experiments to study the performance and usability of WASP: (1) deploy WASP rules into WASP for performance evaluation; (2) usability study where security administrators were asked to configuring WASP rules using RDT. In the first experiment, we configured an environment that consists of two IBM blade servers

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>NewUser</servlet-name>
        <servlet-class>UserManagement.NewUser</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>EditUser</servlet-name>
        <servlet-class>UserManagement.EditUser</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>EditEmail</servlet-name>
        <servlet-class>UserManagement.EditEmail</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>NewUser</servlet-name>
        <url-pattern>/user/register</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>EditUser</servlet-name>
        <url-pattern>/user/update</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>EditEmail</servlet-name>
        <url-pattern>/user/updateEmail</url-pattern>
    </servlet-mapping>
</web-app>
```

Figure 13: XML configuration illustrating three different URLs

connected with high speed LAN (100Mbps). Figure 4.4 shows the environment characteristics, where each server consists of two 2.80GHz Intel CPU with 4GB memory. On S1 server we configured WASP as a plugin inside a IBM Websphere Application Server (WAS), and we also deployed a very simple Web application to receive users input. On S2 server, we installed LoadRunner 8.0 to initiate requests to the web application and monitored response time (RT) in seconds and average Transaction Per-Second (TPS).

We used three types of WASP rules to test performance. All rules used Java-compatible regular expression for defining signatures. The four types of rules are: (1) white list rules which defines positive formats for email, phone number, URL and so on; (2) sanitization rules which translate Web sensitive characters like $<$, %2F,%3C, etc. to correct format; (3) XSS (Cross-Site Scripting) and SQL-I (SQL Injection) rules for protecting against attacks as defined in XSS Cheat Sheet[6] and SQL-I Cheat Sheet,[7] respectively. We also enabled `inheritParent` for all `FilePathNode` instances. We set three different levels of concurrent user: 30 Concurrent User, 50 Concurrent User and 100 Concurrent User.

Figure 4.4 illustrates the performance results. From the results, we can summarize that: (1) WASP rules performs reasonably well in WASF instance; (2) increasing the number of concurrent users can reduce the performance of WASP rules execution; (3) complexity of rules impact the performance of WASP execution; (4) optimizing the processing of regular expression can greatly improve the performance of WASP execution.

In the second experiment, we used the Web application described in Introduction for testing the usability of WASP,

[6] `http://ha.ckers.org/xss.html`
[7] `http://ha.ckers.org/sqlinjection/`

```
"urltemplate":{ "content" :"saas.com/
              myPizzaShop/pizza/{pid}/topping/{tid}",
        "hostname" :
        { "_id_":"H01",
           "content": " saas.com ",
           "filepathnode":[{  _id_":"FPN01",
               "resource":"myPizzaShop",
               filepathnode:[{  _id_ : FPN02 ,
               resource: pizza,
               filepathnode:[{_id_ :FPN03,
                    resource: pid,
                    isVariable: true,
                    variableExpression: ^[0-9]+$,
                    filepathnode :[{  _id_ : FPN04,
                        resource : topping,
                        filepathnode :[{_id_ : FPN05,
                            resource : tid,
                            isVariable: true,
                            variableExpression: ^[0-9]+$,
                        }]
                    }]
               }]
          }]
    }]
}}
```

Figure 14: JSON Representation for URI Template

```
SecRule REQUEST_FILENAME "^/agora/meetings/util
           /publishFile$" "chain,log,auditlog,phase:2"
SecRule ARGS:id "!^[a-zA-Z0-9_]+$"
SecRule ARGS:id "/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)"
SecRule ARGS:email "!^[\\\\w-]+(?:\\\\.[\\\\w-]+
                    )*@(?:[\\\\w-]+\\\\.)+[a-zA-Z]{1,10}$"
```

Figure 15: Rule Sample for ModSecurity

especially of RDT. We used a user management module of a recruiting website that provided Web services on URLs such as `/user/updateEmail`, `/user/updateAddress`, and `/user/updateHobby`. The recruiting services allow users to revise their email, address, hobby, interesting area, career introduction, education background and so on. We invited rule developer to configure white-list rules, XSS and SQL-I rules for each input field. Two rounds of rule configurations were needed: (1) disable rule inheritance; (2) enable rule inheritance. In the experiment, we ignored the complexity of signatures in each rule and regarded that completing each rule binding required one unit of effort. At the end of the experiment, we summarized all efforts used in the two rounds and the results are illustrated in Figure 4.4.

For the usability experiment, (1) when enabling Rule Inheritance (RI), WASP rules reduced the effort to about (18-8)/18=56%; (2) general rules like XSS and SQL_I could be configured as base rule on the root `FilePathNode`, thus all descendent can be protected; (3) with the protection of base rule, a security administrator can focus on configuring rules for specific application logic, for example, EMAIL rule for the service on UpdateEmail.

6 Related Work

In this paper we described the design and implementation of a hierarchical rule schema and a tool for quickly developing firewall rules by exploiting rich semantics of Web applications with out modifying the source code. A number of

```
"RequestMessage": [
    {"method": { "content": "POST" },
     "version":{ "content": ""},
     "URLTemplate":{
     "content": "lotuslivelabs.research.ibm.com/agora/
                 meetings/util/publishFile.jsp",
     "QueryString":{
          "content": "",
          "parameter": [{
             "name": "id",
              binding:[{"rule": "NEG_CHECKXSS" }
                       {"rule": "POS_CHECKALPHANUMERIC"}]
             }]  }} }]

Rule:[{
 "_id_": "NEG_CHECKXSS",
 "name": " CHECKXSSRule",
 "description": "A negative rule for checking XSS attack",
 annotation: Annotation not yet supported,
 condition : {
  description: XSS attack signature,
  value: SIG.MATCH (\"(?:<(?:!\\[cdata\\[|(?:meta|script)
  \\b|(?:base|link| ?:t(?:able|[dh])|body)\\b.*?\\
  bbackground)\\b\\W*?\\b(?:(?:java|vb)script:|.*\\w\\.(?:
  j|cs)s\\s*[\"`?>]))|(?:@import|\\.fromcharcode)\\b|src
  \\b\\W*?\\b(?:(?:java|vb)script:|.*\\w\\.(?:j|cs)s\\s*
  [\"`?>])|\\b(?:style\\b\\W*?=.*?expression\\b\\W*?\\(|
  (?:href|url)\\b\\W*?\\b(?:(?:java|vb)script:|.*\\w\\.
  (?:j|cs)s\\s*[\"`?>])|on[bcdefklmrstu](?:(?<=b)(efore|
  lur)|(?<=c)(hange|lick|ontext|opy)|(?<=d)r(ag|op)|(?<=
  e)(nd|rror)|(?<=f)ocus|(?<=k)ey|(?<=l)o(ad|secapture)|
  (?<=m)(ouse|ove)|(?<=r)esize|(?<=s)(croll|elect|tart|
  ubmit)|(? <=t)ime|(?<=u)nload)[a-z]*?\\b\\W*?(?==)))\"
  ,\"ARG_INPUT\")},
  action : [ {"actionType" : "Block"} ] }]

Rule:[{
  "_id_": "POS_CHECKALPHANUMERIC",
  "name": "CHECKALPHANUMERICRule",
  "description": " Only allows alpha numeric values",
  annotation: Annotation not yet supported,
  condition : {
    description: Non-alphanumeric inputs,
    value: SIG.NONMATCH("^[a-zA-Z0-9_]+$","ARG_INPUT")"
   },
  action : [{"actionType": "Log"},
            {"actionType": "FriendlyErrorMessage
             ('templatedID',
              'Request.QueryString.Parameter["id"]',
              'Checking Alpha and Numeric')"}] }]
```

Figure 16: JSON representation for WASP security rules

```
<action name="agora/*/shares"
       class="com.ibm.agora.MeetingShares">
 <param name="meetingId">{1}</param>
  <result>/agora/api/shares.jsp</result>
    <result name="OK_CREATED" type="http">
     <param name="location">/agora/api/details.jsp</param>
     <param name="status">201</param>
    </result>
</action>
```

Figure 17: Example of Context Information as struts.xml

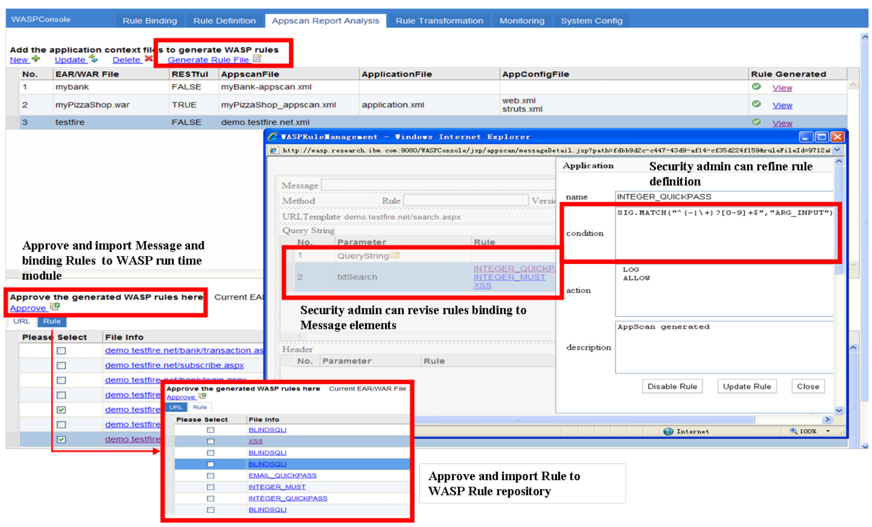

Figure 18: Rule Generation Based on AppScan Report and Application Context Information

vendors have developed in-house WAF, and in this section we first compare and contrast WASP with two open source Web application firewalls: ModSecurity (MS) [8] and Tomcat 7 filter (T7F).[8] Then we will highlight other related work on Web application firewall (we will not compare our work to general network firewall, although one could use such network firewall for filtering Web messages).

MS is a popular Web application firewall (WAF) and is typically installed as an Apache HTTP/Web server module, either in embedded more or in a network mode with Apache reverse proxy server. T7F is a customized Web application server container filter and provides a number of functionality to filters requests and response. T7F is program-based rather than rule-based filter. Both T7F and WASP are written in the Java language, whereas ModSecurity is written in the C language.[9]

MS is probably the most popular WAF and it provides a simple rule language enhanced with support for regular expressions for matching complex messages. Although the MS rule language is simple it is not expressive enough to develop fine-grain hierarchical rules that are needed for modern Web applications that are based on AJAX and REST. MS rules provide limited support for inheriting only the default rules. The semantics of rule chaining in MS restricts how disruptive actions are treated and this restriction complicate composition of rules. For instance, a rule developer cannot simply take two independently developed rules with both containing disruptive actions and chain

[8]http://tomcat.apache.org/tomcat-7.0-doc/config/filter.html

[9]We have also developed a C-version WASP that is also deployed as an Apache module. The C-version WASP does not contain all of the features WASP described in this paper.

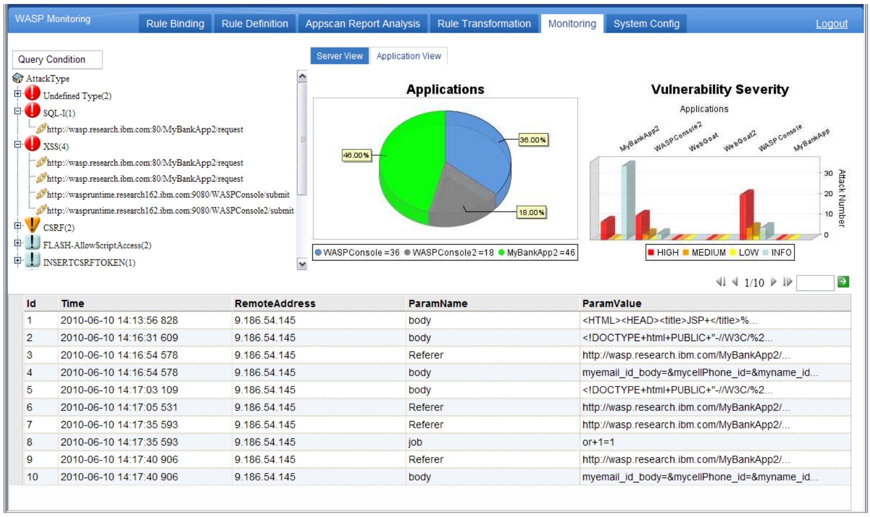

Figure 19: WASP Monitoring Console for Violation Request and Response

them together. MS rules were not explicitly designed to handle RESTful requests, for instance, one cannot easily write rules to validate the URL template variables. MS rule language has been extended using Lua language to increase its expressibility.

Since T7F filters are explicitly coded using the Java language, the concept of rule-chaining does not make sense. One can write customized filters using T7F framework that cannot easily written using a rule language. WASP supports `External Conditions` and `External Plugins` for writing customized conditions and actions using the Java language. Only an expert security administrator should be allowed to use these functionalities, since these external conditions can plugins can themselves become target of an attack. Using the WASP configuration file one can explicitly turn off these features.

WASP also supports a number of other useful features that we have not detailed in this paper. WASP supports a client-friendly feature called SERFS that provide two functionalities: (1) Provide user friendly responses When a request is denied and (2) pre-checking client requests on the client side that we can use to prevent certain kinds attacks. We refer the reviewers to our companion paper for details of SERFS functionality [10]. Both MS and T7F does not provide client-friendly features for end users when firewalls block requests. We have exposed WASP runtime engine to programmatically invoke WASP filtering rules. Consider the following snippet of code.

```
String input = request.getParameter("content");
String output = input;
if(input != null){
IWASPFactory waspFactory = new WASPFactory();
output = waspFactory.processActiveContent(input,
         "HTMLRule.js",IWASPFactory._ContentType_HTML_);
}
```

The above code snippet can be embedded inside a JSP (Java Server Page). The function `waspFactory.processActiveContent(input, IWASPFactory._ContentType_HTML_)` is used to invoke the rules for HMTL content type defined in `HTMLRule.js`. By exposing WASP we have increased the flexibility of WASP for application developer who are not well versed in security to directly invoke WASP functionality so that out-of-the-box default rules are used to protect the application.

Web application firewalls adopt server-side application-level firewall to offer immediate assurance of security. Based on the evaluation criteria of Web application firewall [11],[10] the rule model and management, rule generation, and usability are the key parts to the success of WASF module. Some research work such as Corsaire [11, 4] etc.

[10] `http://www.webappsec.org`

```
GET /agora/meetings/util/publishFile.jsp?id=d44a9d";
</script><script>alert(62415)</script>&load=1 HTTP/1.1
Cookie: com.ibm.bspace.iWidgetEventData=;
bhauth="ZTY3NDQxMTI4MTBmOGU1NWIzNDA1OTVjOTdlNjc4ZTE1OGQ2
NGI3MQ==-MjAxMC0wMS0xMVQxNjoyMzowNS0wNTowMA==-MjAwMDAxMj
Y=-cGthQHVzLmlibS5jb20=-UGF1bGEgQXVzdGVs-MQ==-MjAwMDAxMjE
=-SUJN"; sauidp=U07863540126098465349l;l3sessionkey=qubkq
5aqa9s4dh4fhf8iu09o65; CAKEPHP=lvc6g582c0ft05p2c4hgijqfk0;
JSESSIONID=225A4E4AABFC1B7F1C6AD94400EE21FE
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg,
application/x-shockwaveflash,application/msword,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/xaml+xml, application/vnd.ms-xpsdocument,
application/x-ms-xbap,application/x-ms-application, */*
Referer: https://lotuslivelabs.research.ibm.com/agora/
meetings/host/
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; .NET
CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.0.
04506.648; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Host: lotuslivelabs.research.ibm.com
Connection: Keep-Alive
```

Figure 20: AppScan HTTP message injection example

enable the input validation mechanism in HTTP request phase, to prevent the vulnerability attack based on pre-defined security rules. Also there are some work such as from PaloAlto Network[11], Smask [6], etc., prevent data leakage and attack detection in HTTP response phase. Some existing WASF products such as DataPower[12], ModSecurity [8] etc. define the security policy which can be enforced in the run time engine. DataPower presents the XSD (XML Schema Definition) as the policy syntax implemented in DataPower engine. ModSecurity presents the property text file as the policy syntax implemented in Apache module, and it is planning to extend policy description to Lua script language.[13]

We also studied the firewall rule language defined in DataPower[14], ModSecurity [8], and WAF Language [9] and also studied the UML model defined in the WADL [5] specification. Our HTTP message model defined in HRS supports URLTemplate, which can validate the dynamic input value embedded in dynamic URL, to support more REST message. HRS can express hierarchical fine-grain and semantically rich rules to prevent a broad class of attacks in HTTP request and response. Based on HRS, we implement the WASP rule management tool as a RESTful service[15] to manage the JSON-based rule files. Also a rule transformation engine is implemented in the rule development tool, which can transform the WASP rules to IBM DataPower rules. The WASP rule development tool will contribute to IBM WASF product to enhance the product usability.

Some existing work [3, 2, 1] discuss the virtual patching process for firewall rules in the high level concept. In [1], virtual patching is defined as enabling IT security professionals to regain control over website security by eliminating vulnerabilities as they are detected without developer intervention. The author in [3] discussed the high level work flow for virtual patching process as preparation, identification, analysis, virtual patch creation, implementation/testing, and recovery/follow up. No detail implementation is discussed in these papers. In our paper, we set up four steps to enable the virtual patching for firewall rules. we first create the WASP rule template based on HRS and application context information. From application context information such as web.xml, struts.xml, or WADL file, we can retrieve the URLTemplate especially for the dynamic RESTful URL. Then for the rule generation, we parse the XML report of security testing tool, AppScan[16], to generate WASP message and rules, which will be processed by the runtime

[11] http://bsius.com/media/151160/preventing-data-leaks-at-firewall.pdf
[12] http://www.ibm.com/developerworks/websphere/library/techarticles/0712sheikh/0712sheikh.html
[13] http://lua-users.org/wiki/LuaTutorial
[14] http://www.ibm.com/developerworks/websphere/library/techarticles/0712sheikh/0712sheikh.html
[15] http://incubator.apache.org/wink/
[16] http://www-01.ibm.com/software/awdtools/appscan/

Machine ID	S1	S2
Machine Type	IBM Blade Server	IBM Blade Server
CPU	Intel Xeon 2 CPU 2.80 GHz	Intel Xeon 2 CPU 2.80 GHz
RAM	4GB	4GB
OS	SUSE Linux Enterprise Server 9 (i586)	Windows Server 2003
Middleware	WebSphere Application Server 7.0	LoadRunner 8.0
WASP Deployment Mode	WebSphere Plugin	Not Applicable

Figure 21: Experiment Environment

WASP engine.

Another challenges in WASF product is the usability issue. From this point of view, it is important to bridge the gap between the firewall and back-end applications. For example, we found that when some non-compliant character is detected in server-side firewall, a static error page [8] is throw out to end user, which breaks the consistent of application logic and induces the bad user experience. Some works such as Adaptive Security Dialog [7], JSONR[17]design the programming model in development phase to enable the security feature as well as provide good user experience. The mechanism of smart error reporting is different from these works, which is enabled in runtime firewall engine to protect on boarding applications. It is consistent with the existing application logics and no modification of application codes is required. Instead of throwing out a static error page to end users when some violation is detected in runtime WASP engine, a friendly error message and input backfill mechanism is designed to improve the user experience.

7 Conclusion

In this article, we have designed a WASP rule language and meta-model to handle most aspects of HTTP request and response, including the application context. The application context is often encoded in one or more configuration files and/or annotations that are part of the back-end methods. We can then use the application context to develop a fine-grained and semantically rich WASF rules. Meanwhile, a rule development tool is presented to illustrate how to develop security rules based on WASP rule language. The rule development tool will contribute to IBM WASF product to enhance the product usability.

Virtual patching is a process in which a security administrator will develop and deploy one or more rules on a Web Application Firewall (WASF) to prevent any exploitation of application vulnerability. In this article we discussed the mechanism on how to provision rules using application context and testing tool results for interactive virtual patching. We have built a prototype of our mechanism in the context of a WASP project. Finally we also provided preliminary empirical results focusing on performance of WASP engine for different configuration rules.

Acknowledgement

We wish to thank several people for their support and several enlightening discussions, espcially people from IBM Software Group: Keys Botzum, Bill O'Donnell, Maryann Hondo, Henry Cheng, Steve Ims, and Jason McGee. We

[17]http://laurentszyster.be/jsonr/index.html

Category	30 CU		50 CU		100 CU	
	RT	TPS	RT	TPS	RT	TPS
No WASP	22	1347	51	1150	78	1266
White-List Rules	44	648	75	670	154	658
Sanitization Rules	44	675	76	661	146	692
Anti-XSS/SQL-I Rules	46	647	75	670	159	630

Figure 22: Rule execution in WASP

also thank vendors who helped in the implementations of WASP.

References

[1] Vulnerability assessment plus web application firewall (va+waf). Technical report, F5 Networks and WhiteHat Security Solution, 2008.

[2] Unified web application vulnerability assessment and virtual patching with qualys and imperva. Technical report, Impreva, 2010.

[3] Ryan Barnett. Waf virtual patching challenge: Securing webgoat with modsecurity. Technical report, Breach Security, 2009.

[4] M. Gegick, E. Isakson, and L. Williams. An early testing and defense web application framework for malicious input attacks. Technical report, North Carolina State University, 2006.

[5] Marc J. Hadley. Wadl: Web application description language. Technical report, Sun Microsystems Inc, 2009.

[6] Martin Johns and Christian Beyerlein. Smask: preventing injection attacks in web applications by approximating automatic data/code separation. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 284–291, New York, NY, USA, 2007. ACM.

[7] Frederik Keukelaere, Sachiko Yoshihama, Scott Trent, Yu Zhang, Lin Luo, and Mary Ellen Zurko. Adaptive security dialogs for improved security behavior of users. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part I*, INTERACT '09, pages 510–523, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Ivan Ristic. *ModSecurity Handbook*. Feisty Duck, United Kingdom, 2010.

[9] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th World Wide Web*, 2002.

[10] Vugranam C. Sreedhar, Yu Zhang, Peng Ji, and Shun Yang. User friendly web application firewall with client prechecking. Technical report, IBM Research, 2011.

URL	Required Rules	Rules requiring Tester's effort on binding action		Configuration Efforts In Experiment	
		Disable RI	Enable RI	Disable RI	Enable RI
/user			XSS, SQL-I	0	2
/user/updateEmail	EMAIL, XSS, SQL-I	EMAIL, XSS, SQL-I	EMAIL	3	1
/user/updateAddress	SAFETEXT, XSS, SQL-I	SAFETEXT, XSS, SQL-I	SAFETEXT	3	1
/user/updateHobby	SAFETEXT, XSS, SQL-I	SAFETEXT, XSS, SQL-I	SAFETEXT	3	1
/user/updateInteresting	SAFETEXT, XSS, SQL-I	SAFETEXT, XSS, SQL-I	SAFETEXT	3	1
/user/updateCareer	SAFETEXT, XSS, SQL-I	SAFETEXT, XSS, SQL-I	SAFETEXT	3	1
/user/updateEducation	SAFETEXT, XSS, SQL-I	SAFETEXT, XSS, SQL-I	SAFETEXT	3	1
Total				18	8

Figure 23: Comparison of Configuration Efforts

[11] S. Vries. A modular approach to data validation in web applications. Technical report, 2006.