RC25297 (WAT1207-049) July 23, 2012
Computer Science

IBM Research Report

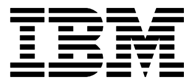
User Friendly Web Application Firewall with Client Pre-Checking

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

Mary Ellen Zurko
IBM Software Group



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

User Friendly Web Application Firewall with Client Pre-checking

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
Mary Ellen Zurko
IBM Software Group

July 23, 2012

Abstract

In this article we present our experience in developing a user friendly Web Application Firewall (WAF) feature, called SERFS, that provide a useful and friendly feedback to the end user when one or more requests are filtered or blocked by the WAF. SERFS provides appropriate friendly templates and rules to make the client/server interactions as transparent as possible. When dealing with forms, SERFS will back fill legitimate values and provide hints to illegal values. An application rule developer can either use SERFS's friendly templates or develop new templates to ensure that the usable security mechanisms are consistent with existing application logic. To improve the response time SERFS also issues JavaScript components to the client browser for pre-checking client requests. If a pre-checked request still contains non-compliant requests, SERFS will identify them as being suspicious and perform further analysis to either block the request or ask the end user to re-authenticate the session. We have implemented SERFS, and we present preliminary empirical results to validate our user friendly features.

1 Introduction

A Web Application Firewall (WAF) is often used to provide a transparent protection and immediate patching for back-end Web applications when a vulnerability is exposed. A WAF should prevent suspicious requests and allow legitimate requests. Unfortunately, a WAF is not perfect in discriminating between suspicious and legitimate requests, which means that it will sometimes allow suspicious requests and deny legitimate requests. When a request is denied often a WAF will throw one of the standard HTTP error pages [11], such as `500 internal server error`. Typically when a back-end Web application processes bad requests, it will provide a more user friendly and possibly customized response for such requests, rather than throw a default error page. User experience is extremely important for Web application in order to attract good end users to visit the Web pages of a Web application.

There are several key challenges for ensuring that a WAF is user friendly: (1) How to provide a friendly error message that is transparent to the end user and the semantics of the error message should be close to what a back-end application would respond? (2) How to develop user friendly response features without unduly burdening the rule developer to implement complex mechanism for customizing friendly responses? (3) How to ensure that the filtering rules does not compromise the security of the application? In other words, the WAF filtering rules must not expose security vulnerability details to the end user for which the application might need to keep obscured. (4) Finally, how to ensure that the response time of error messages does not affect the end user experience?

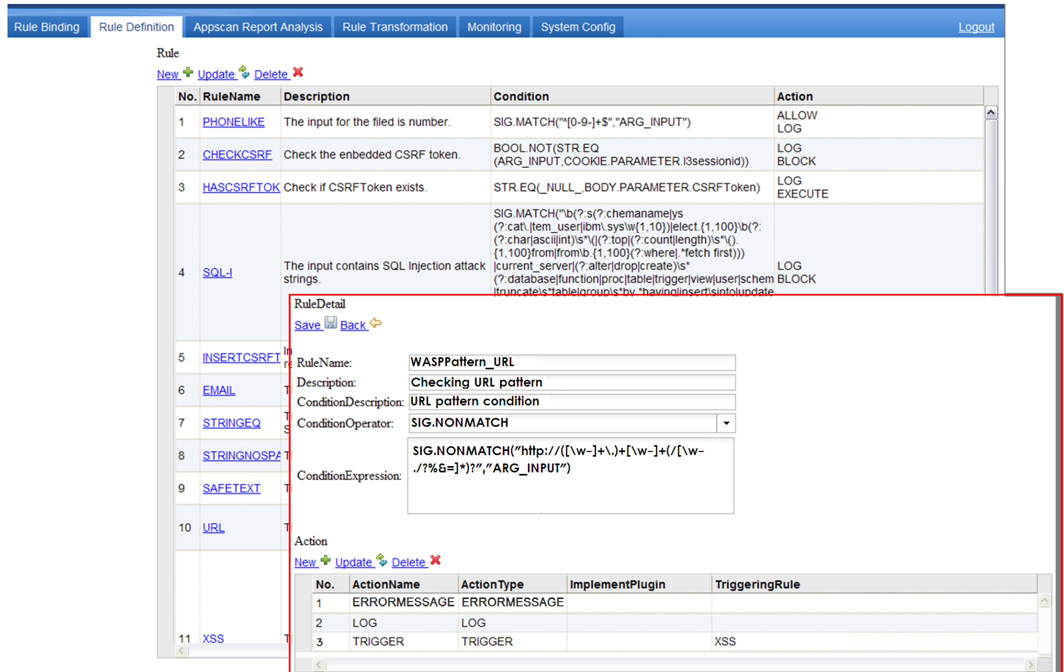


Figure 1: Rule development tool in WASP

It is important to keep in mind that there two ways to think about user friendliness. First is the technical dimension of mapping error message to a specific context so that the user can recognize, diagnose and recover from the error. Second is the user experience dimension by ensuring that the user continues with his/her original process, e.g., filling in a form. By transparent we mean that the user is oblivious as to whether the message came from firewall or the back-end application, and also the WAF filtering rules must not expose security details to the end user.

In this article we describe the design and implementation of user friendly features of a WAF, called WASP (Web Application Security Protector), that addresses the above challenges. Our user friendly feature, called SERFS (Smart Error Response Feedback Service), does not require any modification to back-end Web application. SERFS is based on three design principles: (1) user friendly responses should be easy to develop and should be customizable, (2) user friendly responses should be easy to deploy and enabled, and (3) user friendly responses should not degrade the performance of a WAF.

SERFS contains two main parts:

- The user friendly response feature is typically initiated when a client enters a session with a back-end application, such as when a login page is visited. The user friendly feature is enabled by first issuing a stand alone AJAX-based JavaScript (JS) component to a client browser. The JS component is then used to retrieve the friendly templates from the server-side WASP periodically, and the templates are then stored in the browser cache. When some violation is detected by the WASP an error message with specified format is returned to the client browser. The error message is then parsed and processed by the issued client-side JS component. The JS component then generates the user-friendly security response and back fills user's legitimate inputs of the application forms that was previously entered by the user. The backfilling is needed to ensure that the user experience is not compromised and that the user can continue his form filling process. An interesting aspect of our SERFS design is that an application developer who wants to patch a vulnerability using WASP can customize the template and error messaging to be consistent with the existing application logic and client-side logic without modifying the back-end application.

- To improve the performance when enabling the user friendly response in WASP the SERFS.js components will perform a number of pre-checking and partial validations on the client side. If a request is pre-checked

```
"Rule":[{
"_id_": "R01",
"name": "Rule01",
"ruledescription": "Checking URL pattern",
"condition" : {
"conditiondescription": "URL pattern condition",
"value" :
"REGEX.NONMATCH(WASPPattern_URL, Request.
                  Body.Parameter["homepage"])" },
 "action" : [{"actionType" :
 "FriendlyErrorMessage('templatedID','Request.
  Body.Parameter["homepage"]','Checking URL Pattern')"}]
}]
```

Figure 2: An example of a rule in JSON

and the request still contains suspicious pattern, it will be tagged for further analysis by the WASP engine. In SERFS design we ensure that the client pre-checking and validation does not compromise the security of WASP back-end checking. The pre-checking helps in identifying malicious users from friendly users who may enter non-compliant values in forms due to carelessness. One thing we ensure that if an attacker tampers with the information used at the client side then that information is invalidated when it is sent to the server side application.

We have implemented SERFS in WASP and deployed in Apache server with reverse proxy.[1] We will report some initial user experience and also evaluate its user friendly features and the performance. The rest of the article is organized as follows: Section 2 presents the overview design of SERFS and the WASP rule language and development tool. Section 3 describes the detail design of user friendly response in SERFS. Section 4 describes the design of pre-checking mechanism in SERFS. The user experience and performance evaluation experiment in SERFS are presented in Section 5. The related work is discussed in Section 6. We will conclude our work in Section 7.

2 SERFS Overview

In this section we give an overview of SERFS and also highlight some of the key components of SERFS. The development of user friendly response feature in SERFS consists of the following key steps:

- A rule developer will use WASP Rule Development Tool (RDT) and develop one or more security rules with user friendly response enabled (Figure 1). An application rule developer can also customize the user friendly response or develop new template that is consistent with back-end application error response. We will show how to back fill legitimate form values and flag illegal form values.

- When a client user visits the login page of an application, SERFS-enabled WASP injects an AJAX-based JavaScript component (SERFS.js) along with the response page of the Web application. In this article we assume that the back-end application response page is a form that the end user will fill and submit back to the back-end application.[2]

- The SERFS.js then interacts back with the WASP server engine to retrieve other useful information such as templates for rendering, pre-checking component, etc.

[1] We have both C language version and J2EE complaint WASP implementation. In this article we used C language version and deployed as an Apache module.

[2] This would be a challenge for mobile application, and we have not addressed mobile application challenges as yet.

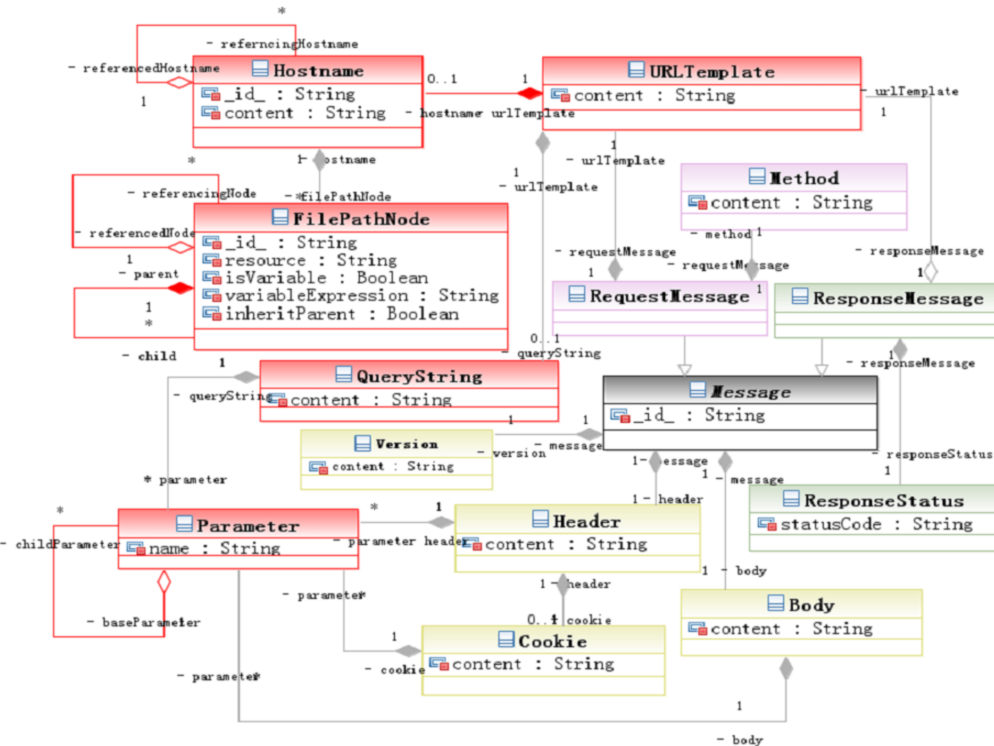


Figure 3: HTTP Message Model

- When the client submits the form back, SERFS.js will interact with WASP engine to enable user friendly response when some violation is detected by the WASP engine.

The design and implementation of WASP is based on two key concepts: (1) hierarchical and fine-grain rule schema with first-class support for URI (Uniform Resource Indicator) templates and REST; and (2) a flexible RDT and scalable rule execution engine with user friendly responses.

2.1 Hierarchical Rule Schema

We have designed and implemented a fine-grain hierarchical rule schema for WASP to address the problem of fine-grain protection of Web application without modifying the vulnerable application and preventing loss of any its functionality. The HRS is based on two design principles: (1) Ability to support fine-grain rules to protect hierarchical Web resources and services and (2) Late binding of rules to message types. A typical Web applications based on AJAX and REST maintain a large collection of hierarchical resources and services. HRS and rule instances are implemented and stored as JSON (JavaScript Object Notation) objects and the rules are managed using REST (Representation State Transfer) services. The JSON representation allows one to enrich the rule schema to include meta-data and other information. To enable usability for application rule developer, we have seamlessly integrated customizable friendly responses in our HRS.

The HRS consists of three main parts: (1) HTTP Message Model that define the core model of HTTP message structure (see Figure 3). The HTTP message model identifies all of the elements of HTTP request and response messages and their relationships. Our HTTP message model is tailored for writing WAF rules. (2) Rule Model that can be used for writing firewall rules (see Figure 4). A simple firewall rule is of the form if (condition) then (action), where condition is one or more constraints on instances of HTTP message model and action is what the security

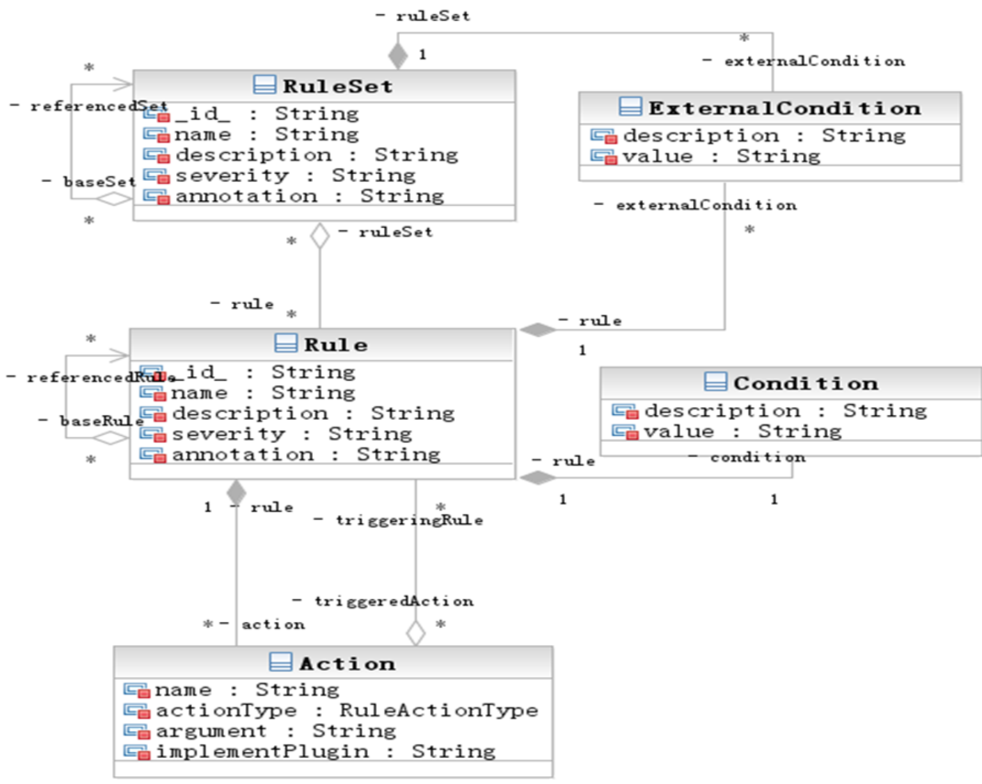


Figure 4: Rule and Rule Set Model

administrator will want to take when the condition is satisfied. (3) Message Rule Binding is necessary to determine the set of rules that should be triggered at runtime for a given message. The following is an example of a rule:

```
R01: if (REGEX.NONMATCH(WASPPattern_URL,
      Request.Body.Parameter["homepage"]))
   then Action.FriendlyErrorMessage (templatedID,
        Request.Body.Parameter["homepage"],
        "Checking URL pattern")
```

The JSON representation of the above rule is illustrated in Figure 4. The HTTP message model is used to identify key elements of a HTTP request. For instance, in the above rule `Request.Body.Parameter ["homepage"]` identifies a particular “body parameter” and checks if its value matches the regex `WASPPattern_URL`. The *parameter* element is a name-value pair, and in the above rule, the name is `homepage` and `Parameter["homepage"]` returns its value. If a user’s input value does not match the `WASPPattern_URL`, a friendly error message is returned with friendly `templateID` to the client based on the definition of `Action.FriendlyErrorMessage()`. The parameters of the `FriendlyErrorMessage` are needed for enabling friendly response, and will be explained in detail later. The action `FriendlyErrorMessage()` is also a hint to the WASP engine to enable SERFS friendly response, and at this point, if SERFS.js is not cached in client browser side, the WASP engine send this JS component to the client browser.

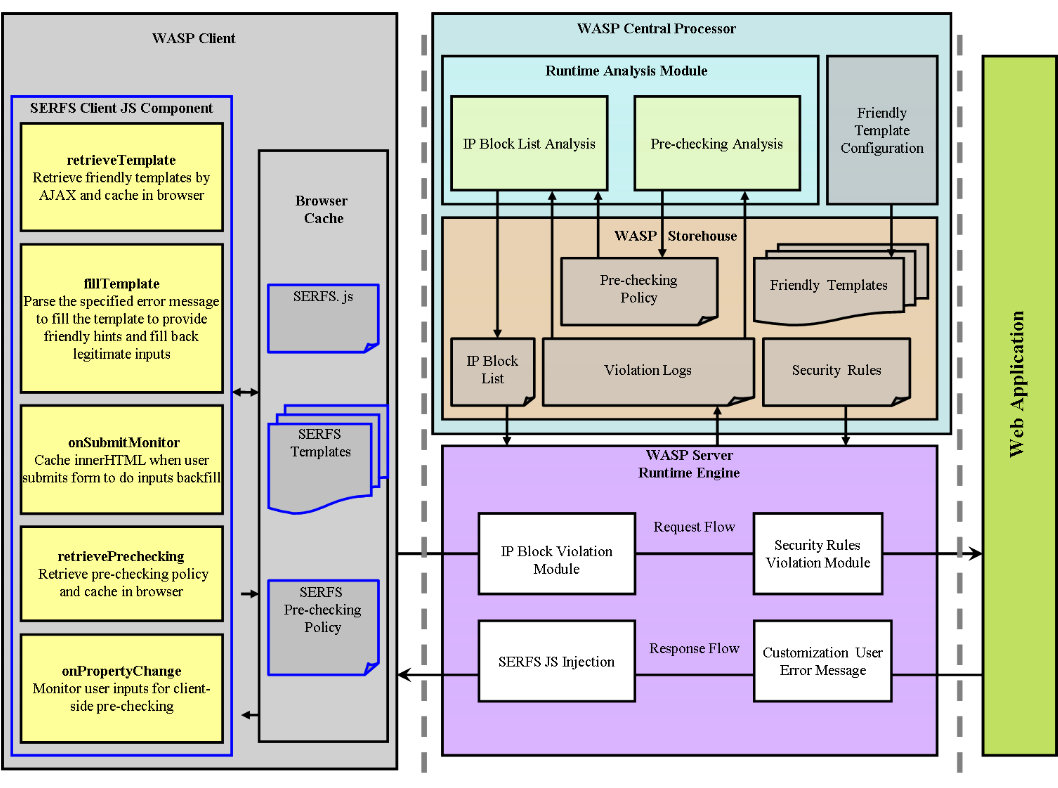


Figure 5: Key components of SERFS

2.2 WASP and SERFS Components

Figure 5 shows the key components of WASP for enabling SERFS. WASP consists of three main parts: (1) SERFS enabled WASP client, (2) SERFS enabled WASP server, and (3) WASP central processor. In this article we mainly focus on those parts of WASP that are relevant for enabling SERFS. The WASP Store House contains information that are needed for enabling rules, such as security rules developed by rule developers, violation logs that can used for analysis, and pre-checking policy that can be used for pre-checking user requests on the client side (to be explained later in the article). The WASP runtime engine executes rules and performs other on-line activities, such as sending SERFS.js component to the client side. The runtime engine checks each request based on the *conditions* in the pre-defined rules and will trigger the *actions* if some violation is detected. It will then construct an error message with friendly template ID based on the rules and send it back to the client browser. The runtime engine will insert a stand-alone SERFS JavaScript component (SERFS.js) into the application response page when the user initially logs in to the application. The dispatched SERFS.js component enables client-side functionalities that are needed for user friendly response (see the next section). The functionalities include retrieving friendly templates from WASP periodically, capturing and caching the input values of users, etc. When some violation is detected on the server-side WASP the specified error message from runtime engine will be returned to SERFS.js. A user friendly error message will be rendered, and where possible legitimate user input values will be back filled. Furthermore, based on the violation statistic, partial validation logic in WASP will be dispatched to the SERFS.js component to do pre-checking. This can help users to reduce mistakes when filling the form. If a pre-checked request still contains non-compliant content, it will be identified as a possible attack by the runtime engine and the violation is logged. The runtime analysis module can then use the logged information for further processing, such as to update the IP block list or to re-authenticate the user session. It is important to remember that the rendering of user friendly responses and back filling of legitimate values are done by WASP, without modifying any of the back-end applications. Interestingly, Scott and Sharp point out that Web application vulnerabilities are inherent in the Web application code and is independent

```
{"ID":"090601","name":"overlay\_mode",
"content":"<div class='flash success' id='flash'>
<a href='javascript;'
    onclick='SERFS_hide($('flash'));
   'class='close'>x</a>
<%ruledescription%>,<%violationFieldName%></div>"}
```

Figure 6: A example of SERFS template in JSON for friendly response

```
<script src="http://wasp.research.com/SERFS.js"\>
<script> SERFS_fillTemplate(09060001,/*template ID*/
 "Checking URL pattern",/* rule description */
 "homepage")/* parameter that violated a rule */
</script>
```

Figure 7: Error message from WASP that includes the SERFS.js and invokes `fillTemplate` function

of host and the network technologies [12]. So our pre-checking rules and the corresponding firewall rules is agnostic to network and host technologies.[3]

3 User Friendly Response

The user friendly response features of SERFS consists of two main parts: (1) design of friendly templates and error messages that are needed to render the friendly response when a violation of a rule occurs, and (2) design of SERFS.js component that is dispatched to a client browser and cached in the browser.[4] SERFS.js component has a number of functionalities including retrieving the friendly template from server-side WASP and back filling forms with legal values of the form previously entered by the end user, etc.

3.1 Friendly Templates and Error message

SERFS provides support for rendering a message in a more meaningful and esthetic manner using pre-defined and customizable templates and error message formats. Figures 12 to 15 illustrates a few examples of the rendering templates that are available in SERFS. An application rule developer can customize the style of friendly templates based on the requirements of different applications. The template files can be cached in the browser to improve the performance. We use JSON representation to model templates and each template consists of a template `ID`, template `name`, and template `content`. A simple example of SERFS template using JSON is shown in Figure 6.

When a WASP rule containing user friendly actions such as `Action.FriendlyErrorMessage()` is enabled, a friendly error message with the specified format is returned to the client browser. The error message consists of a JavaScript function (e.g., `SERFS_fillTemplate`) that includes the template ID, rule description and violation parameter (Figure 7). Once an error message is sent to the client browser, it will trigger the function in SERFS.js component to retrieve the template from browser cache and render the friendly response in the client browser.

[3]Another aspect that we have not addressed here is that how pre-checking rules might impact and interact with automated vulnerability testing, or even bugs in the client side code. This is a topic for future research.

[4]Although we have not implemented globalization for error messages, this can easily be added in our implementation to render error message in different languages using either cookies or user preferences.

```
var SERFS = document.createElement('script');
SERFS.setAttribute('src',
'http://wasp.research.com/SERFS.js');
document.body.appendChild(SERFS);
```

Figure 8: SERFS JS element inserted to Web pages

Table 1: Functions of SERFS JS component

retriveTemplates	Retrieve the friendly templates by AJAX method from server side periodically, and cache the templates in browser
fillTemplate	Parse the error message from WASP when some violation is detected, retrieve the friendly template from cache based on the template ID, fill the template with the violation content
fillBackLegitInputs	Fill back the legitimate input values into the browser document
onSubmitMonitor	Capture and cache the input values when the user submits the form to the back-end application
retrivePrechecking	Retrieve the pre-checking policy by AJAX method from server side periodically, and cache the pre-checking policy in browser
onProperty ChangeMonitor	Monitor the user input based on the pre-checking policy to help user reduce to input non-compliant content

3.2 Design of SERFS JavaScript Component

In this section we describe the key functionalities of SERFS.js component that is sent to a client browser. Figure 5 and Table 1 illustrates the key functions of SERFS.js on the client side to enable friendly response with back-fill mechanism. The back-fill mechanism is needed to automatically fill in the legitimate entries of the form; and the friendly error message is rendered for illegal form values. A detailed message sequence of the SERFS.js component and the server-side WASP engine is illustrated in Figure 9, and it includes the following steps:

Step 1: The SERFS.js is injected into the application response page by the WASP engine when a user is first presented with a login page.[5]

Step 2: SERFS.js is cached in client browser. Several functions of SERFS.js shown in Table 1 are added to the browser document.

Step 3-5: The AJAX function `retriveTemplates` retrieves the appropriate friendly templates and caches them in browser.

Step 6-7: When a user fills in the application form and submit the request to Web applications, the `onSubmitMonitor` event is invoked to capture the user inputs. The user inputs are cached in the browser memory for subsequent back filling of legal values.

Step 8: The user request is then intercepted by server-side WASP engine, and it is validated based on the security rules. If no violation is detected, the request will be forwarded to the back-end applications, otherwise an error message, as shown in Figure 7 with specified template format will be returned by the WASP engine, and the client-side SERFS will render the error message on the client browser.

Step 9-11: The `fillTemplate` in client side will parse the error message that was sent by the WASP server, retrieves the template from browser cache, backfills the legitimate input values of the user in the browser document and then renders the friendly error message. Note that the legitimate back fill values are stored in the browser session memory, and our SERFS.js will use these values to back fill with legitimate values.

[5] SERFS.js can also be injected with other pages too, depending on WASP configuration.

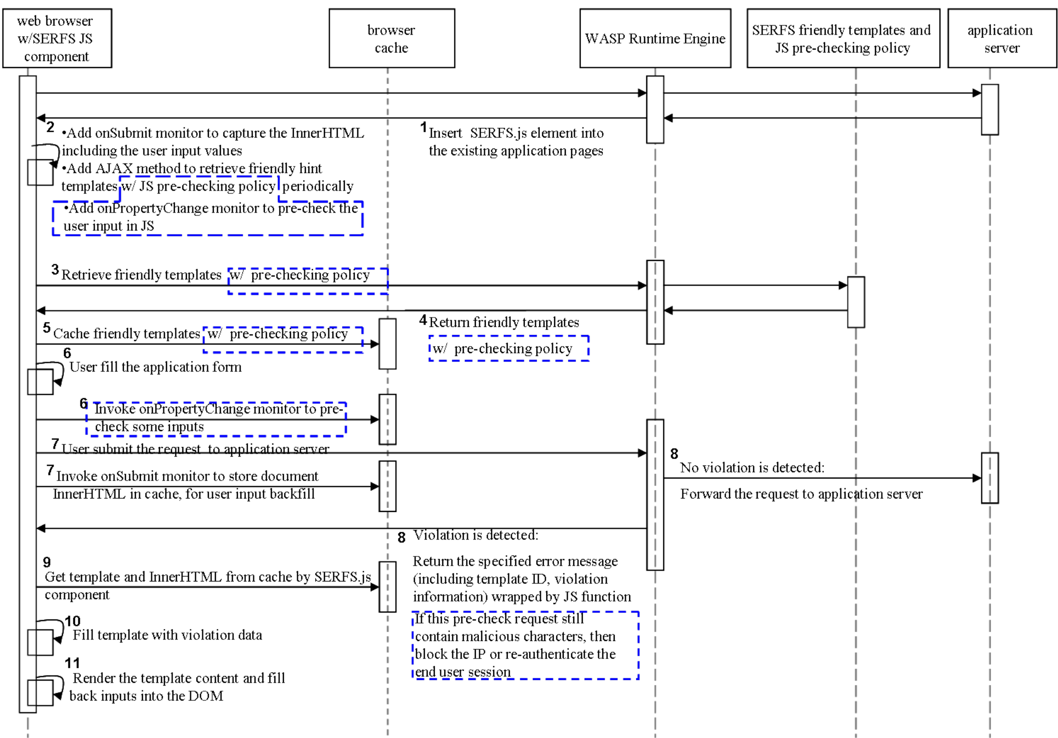


Figure 9: Sequences to provide user-friendly error reporting, input backfill and pre-checking in SERFS

4 Pre-Checking Mechanism

To improve the performance when enabling user friendly response in WASP we implemented partial validation logic in the SERFS.js component to enable client-side pre-checking. This mechanism can help legitimate users to reduce "typo errors" when forms are filled. It can also block suspicious users if a request that should have been pre-checked is sent to the WASP engine with illegal form values. An attacker, who intercept the user request and modify the form content will typically bypass the pre-checking logic that validate the requests.[6]. In such cases WASP pre-checking analysis module (see Figure 5) can either re-authenticate the user session or even block IP from such requests. It is important to keep in mind that IP blocking has limitations. In some cases, because of outgoing proxies, entire organizations might seem to be coming from one IP address.

The pre-checking policy and the validation logic that is issued to client browser represented using JSON format and is cached in client browser (as shown in Figure 10). For example, the input field named as "homepage" in the form "profile" should match the regex pattern "`http://([\w-]+\.)+[\w-]+ (/[\w- ./?%&=]*)?`"; the input field named as "address" in the form "profile" should not contain the risky character defined in the regex pattern "`<>='$\"\;!--`". To enable such client-side pre-checking, we add two additional functions in the SERFS.js component as shown in Table 1, for retrieving the pre-checking policy and monitoring the `onPropertyChange` event to validate the user inputs based on the pre-checking policy.

The pre-checking analysis module shown in Figure 5 is used to update the pre-checking policy on-the-fly based on the statistic of the violation logs detected in WASP. For instance, if the violation ratio for the input field "homepage" is higher ,than the predefined threshold value, the regex pattern to validate the "homepage" field can be used to updated the pre-checking policy and the policy is then issued to client side. For security reasons not all rules will be pre-checked by the client-side SERFS.js. For instance, the pre-checking rule may not have been developed and deployed until the violation ratio exceeds a threshold for certain filtering rules. For security reason if the server side rules were insufficient, they could be fully analyzed more easily and rapidly by an attacker. Or if one site was covering up a

[6] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

```
{"ID":"090701","field":"email","operator": "match",
"pattern":"http://([\w-]+\.)+[\w-]+
   (/[w- ./?%&=]*)?", "form":"profile"},
{"ID":"090702","field":"address","operator":"nonmatch",
"pattern":"<>=’$\"\;!--","form": "profile"}
```

Figure 10: SERFSpre-checking policy

vulnerability in an application also run by another site, which did not know about the exposure. In addition, they provide little to no utility to a ”good” end user; they would just catch ”cut and paste” attacks client side.

It is important to keep in mind we can indeed pre-check any invalid URL pattern on the client-side, but for security reasons, we have limited pre-checking patterns to those that a legitimate users would make inadvertent mistakes, rather than an attacker sending malicious URLs. Also, if the pre-checking logic includes complex patterns, then an attacker can learn those patterns; and so we perform complex checking on the server-side.

Instead, the SERFS engine will collect the user input values which trigger the negative pattern often, such as “`<script>`” and “`javascript`”, to update to the pre-checking policy, and then do simple string pattern matching in client-side SERFS.js component for pre-checking. The corresponding operation sequences between the SERFS.js component and server-side WASP engine are updated in Figure 9 and marked with dashed box. The updated operation sequence consists of:

Update for Step 2-5: The SERFS pre-checking policy will be retrieved together with friendly templates using the AJAX method in the JS component, and then cached in the client browser.

Update for Step 6: When a user fills the application form, the `onPropertyChange` event will be triggered to check the user inputs based on the pre-checking policy.

Update for Step 8: When a pre-checked request that is sent back to WASP engine still contains non-compliant content, the client IP address will be identified as suspicious and further analysis will be done by the runtime analysis module to either block the IP or re-authenticate the user session. It is important to note that a legitimate pre-checked request can be modified by a man-in-the-middle attacker and our WASP run time analysis will verify if the request still contains illegal values and mark it as possibly being malicious.

5 Experiment Evaluation

In this section we describe our experience with SERFS, focusing on user experience and performance evaluation.

5.1 User Experience Evaluation

Even though there are several well published techniques for evaluating user experience, it is often difficult to get accurate result. There are many ways to evaluate user experience, and each has its place in the lifecycle. We selected heuristic evaluation by experts in order to get early feedback on the broader user experience issues. For our purpose we chose two of the ten Nielson’s heuristic metric for evaluating SERFS user experience [9].

- **User control and freedom:** This heuristic is based on the observation that users often make mistakes and want clearly marked “exit” that allow them to leave without going through unwanted dialogue.
- **Help users recognize, diagnose, and recover from errors:** Error messages should be expressed in plain language that precisely indicate the problem. It should also provide useful suggestion for addressing the problem.

We borrowed concepts based on the above two Nielsen’s metrics to design our survey questionnaires and also since they highlight two key features of SERFS: (1) legitimate end users often make simple errors, such as typing errors when filling forms and so a firewall should not penalize such errors, and (2) when such mistakes are made end users expect that the legitimate form values are correctly back filled and friendly responses are provided for the errors.

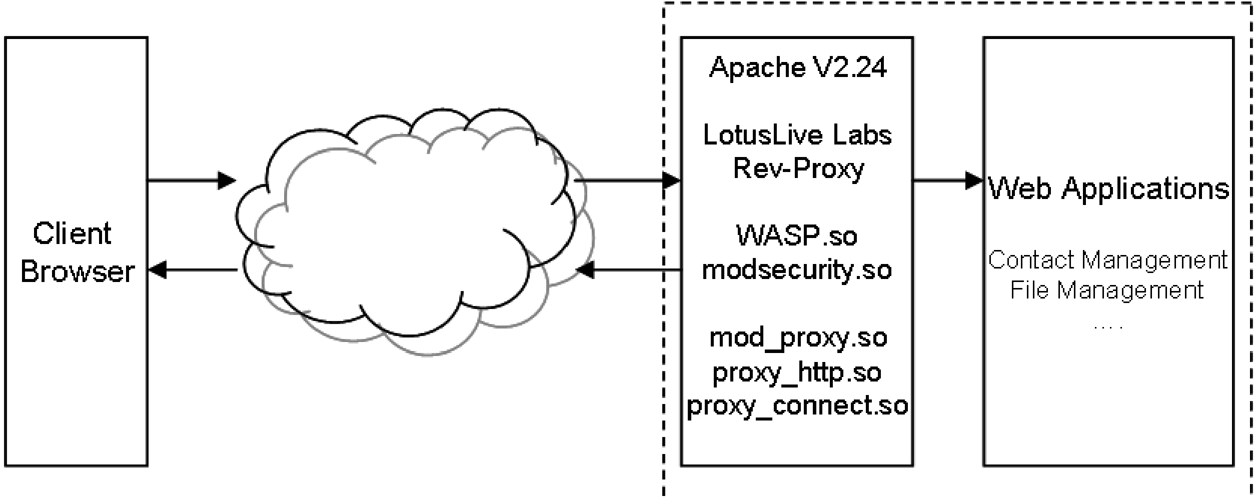


Figure 11: WASP and ModSecurity deployment in Apache server

To evaluate SERFS we deployed WASP in a Apache server and also enabled Apache reverse proxy functionality as illustrated in Figure 11. We used the applications in LotusLive Labs as the demo applications to evaluate SERFS, that is accessed through the Apache proxy server. The HTTP traffic is validated by WASP engine. The SERFS friendly templates and JS component are dispatched to client browser by WASP to enable the friendly error message with back-fill mechanism. To compare the user experience of SERFS with another WAF, we also set up the open source ModSecurity WAF in our experiment [11]. We set up the following three scenarios for evaluation:
Scenario 1: Deploy ModSecurity in Apache Web server to provide security protection. When some violation is detected, a static error page `500 Internal Server Error` is returned to clients (see Figure 12). Even though this scenario seems unfriendly response, we often find that many Web site do indeed provide such common response for error message due to convenience or to hide obscurity of of Web application logic.
Scenario 2: Deploy WASP in Apache Web server and enable the friendly hints to clients. When some violation is detected in WASP engine provide friendly response (see Figures 13 and 14).
Scenario 3: Deploy WASP in Apache Web Server and enable the back-fill mechanism to maintain the legitimate input value when some violation is detected in WASP engine (see Figure 15).

To evaluate the two Nielsen's heuristic metric we invite 11 tester who have Web development, user experience and testing background. We designed a simple set of survey questions such as:

```
  Does the system provide sufficient information to explain the error?
  Does the system provide sufficient guidance to help you fix the error?
  Does the loss of content caused by ill-formed input affect the whole
process?
  What is your user experience for re-entering the content after fixing the
error"
```

For each question we asked the user to provide a rating from 1 to 5, with zero indicating a highly negative experience and 5 indicating a highly satisfied positive expereince. For the first evaluation metric, the mechanism of SERFS friendly response we got on average of 3.6 value for SERFS compared to 1.5 for ModSecurity experience, indicating that most of the users for SERFS were highly satified when compared to ModSsecurity experience. For the second evaluation metric, the back fill mechanism that maintain the legitimate inputs with SERFS experience was on average 3.4 where as for ModSecurity we obtained an average score of 1.2, once again indicating that the user experience is improved with the friendly error message and backfill mechanism in SERFS. To limit user experience bias, we selected half of the users to first use ModSecurity and then use SERFS, and the second half of the users to use SERFS first and then ModSecurity.

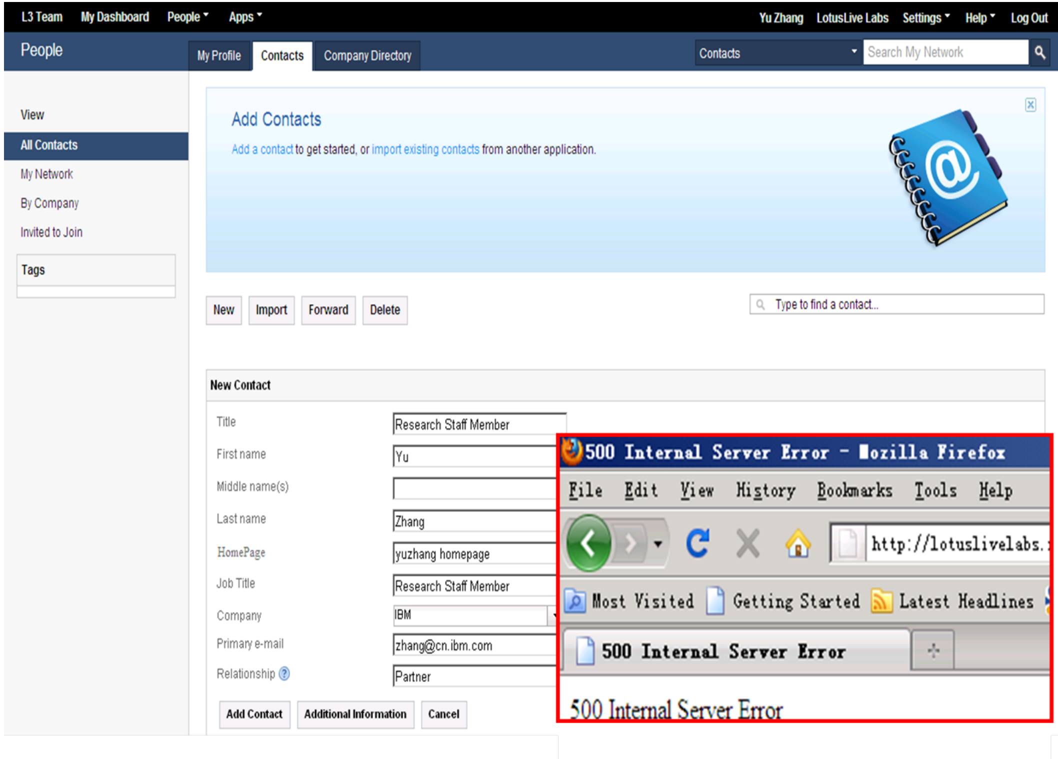


Figure 12: Static error page

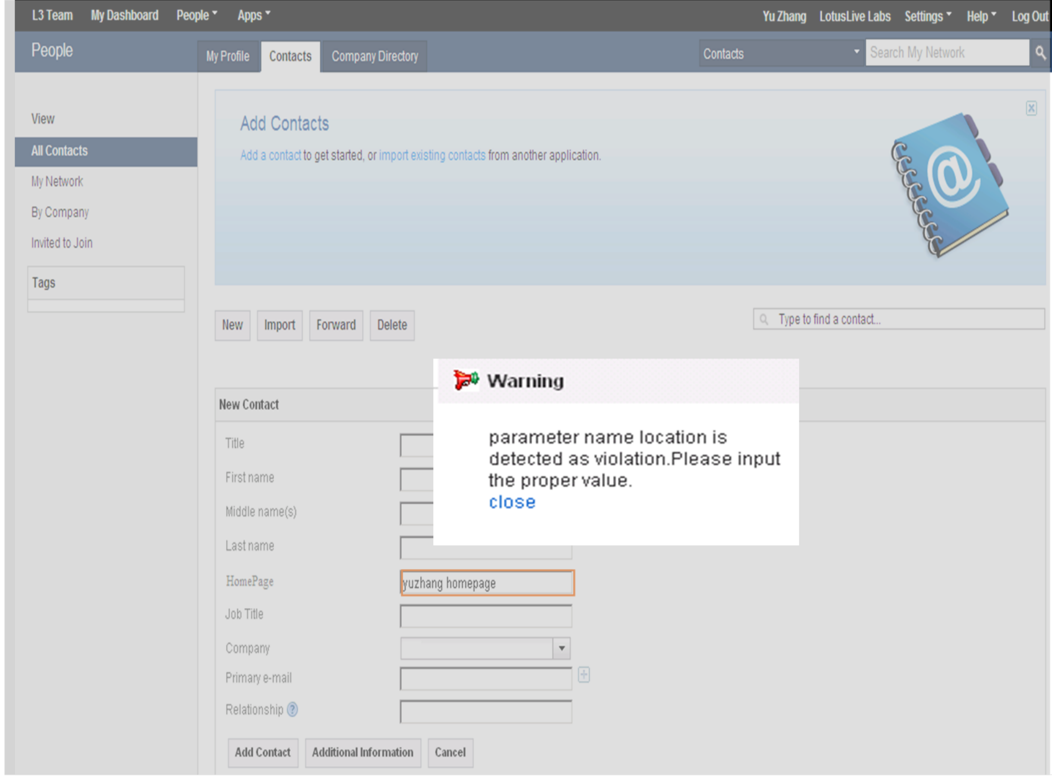


Figure 13: Friendly error message with overlay style

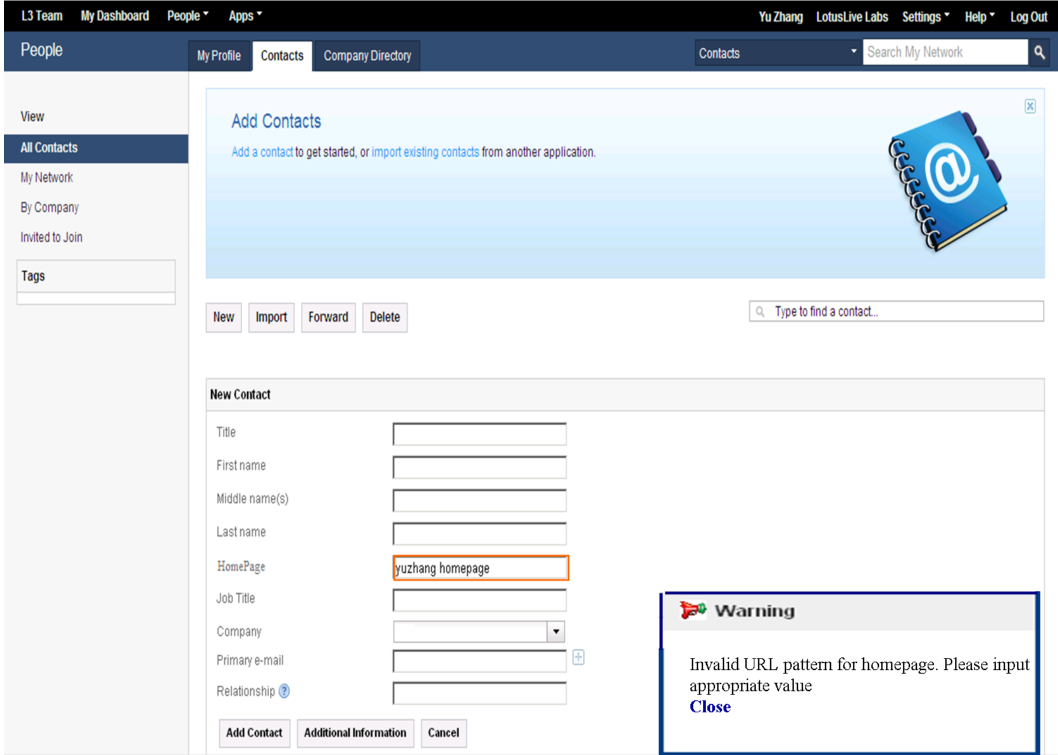


Figure 14: Friendly error message with hints style

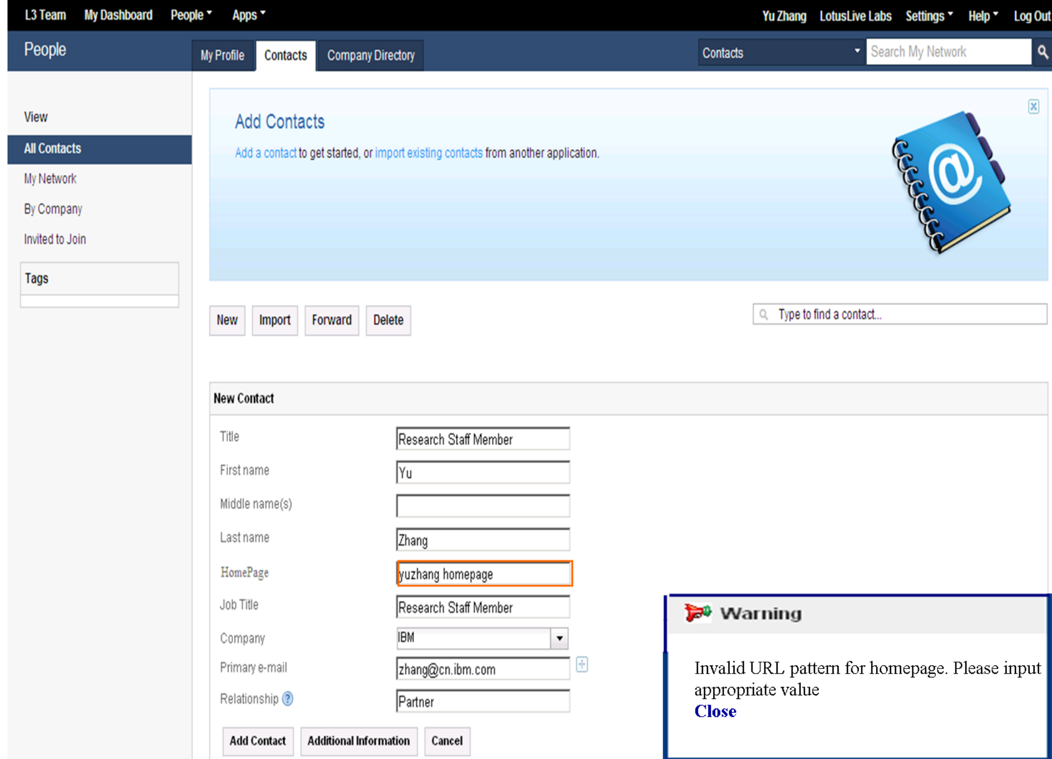


Figure 15: Friendly error message with hints and backfill

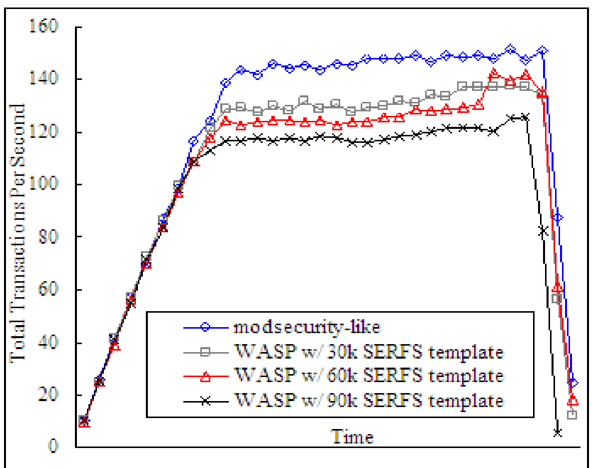


Figure 16: Total transactions Vs. different sizes of SERFS templates

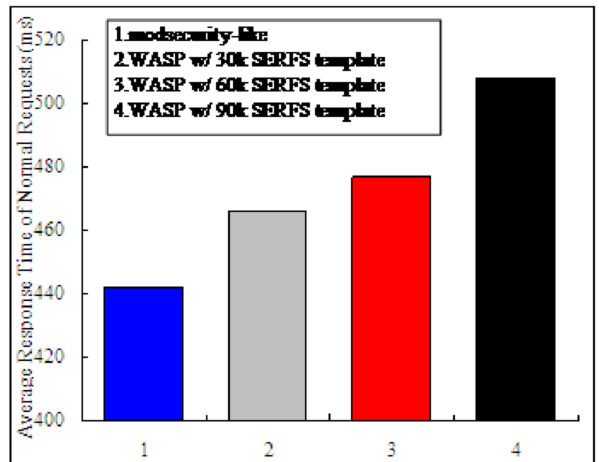


Figure 17: Average response time Vs. different sizes of SERFS templates

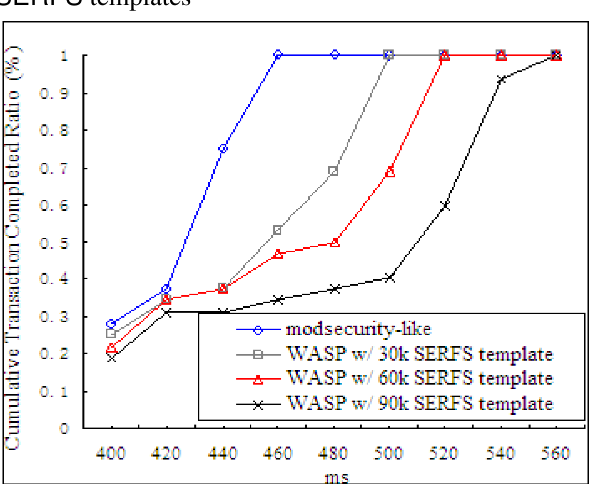


Figure 18: Cumulative transaction completed ratio Vs. different sizes of SERFS templates

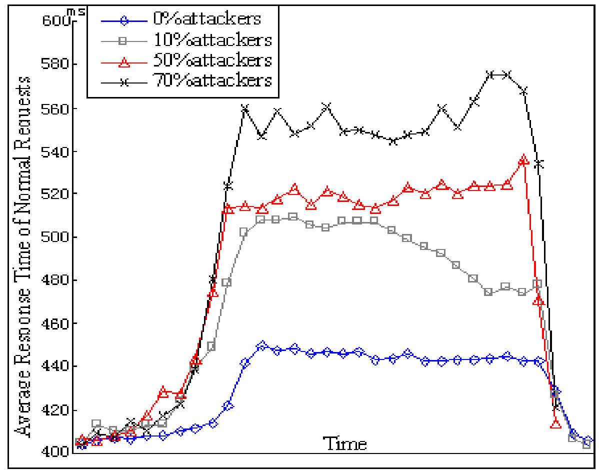


Figure 19: Average response time for different attack ratios

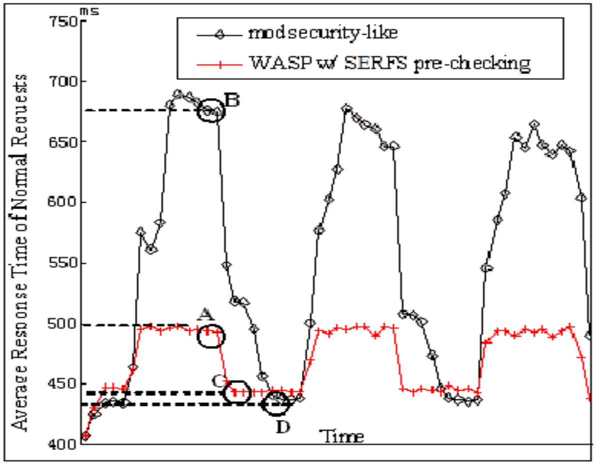


Figure 20: Average response time for ModSecurity and WASP with SERFS pre-checking

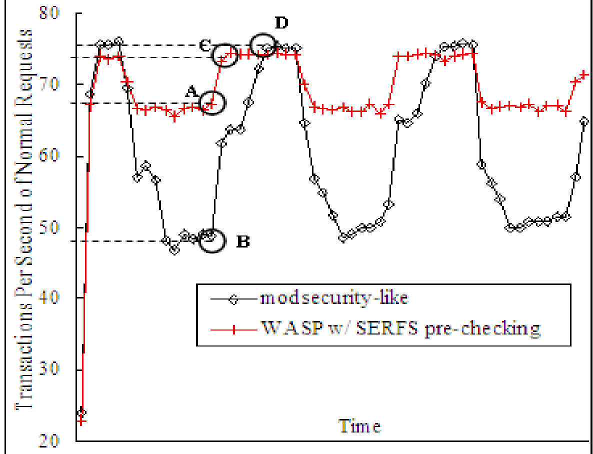


Figure 21: Transactions for ModSecurity and WASP with SERFS pre-checking

5.2 Performance Evaluation

To evaluate the performance impact of SERFS, we conducted our experiments on a 4-node machine connected via a high-speed LAN. All of the nodes are Intel Core2 6700 2.66GHz, 3G RAM machines with Window XP professional operating system. The nodes are connected to the Internet through a 1000M bps connection. One node is deployed with LoadRunner (version 8.0) and mimics client browsers, simulating multiple users by sending concurrent HTTP requests to server application. Two of the nodes are used on the server side to process HTTP requests. WASP is deployed on one of the nodes in the Apache server (version 2.24); and the other node is deployed with a Tomcat server (version 5.5) and IBM DB2 server (version 9.0). SERFS analysis module (see Figure 5) is deployed on the fourth node to update the pre-checking policy and IP blocking policy by analyzing the violation logs from WASP engine. The fourth node is also used to update the SERFS templates for friendly report to end users. No other tasks were running on each node.

5.2.1 Different Sizes of Friendly Templates

To evaluate the performance of SERFS, we used LoadRunner to generate 4,000 bytes of HTTP POST requests with 6 different parameters. One of the parameters contains a large *textarea* for users to input comments. For WASP runtime engine in Apache server, the initial configuration for WASP validation rules is set up with 148 rules in total, containing 55 positive rules (such as the regex pattern for ”EMAIL”, ”URL”, etc.) and 93 negative rules (such as the regex pattern for ”XSS”, ”SQL-I”, etc.). These sets of positive and negative are a snap shot that we collected while implementing and testing WASP.

The pattern matching cost for the negative security rules is typically higher than for the positive security rules due to longer and more complex signatures in the negative rules. For each request, we first validate with positive security rule, and if it fails the request is then validated with negative rules. A user friendly error message is generated and returned to client if the security rule blocks the request. We deployed a simple Web application in the Tomcat container to verify the security protection of WASP runtime engine. SERFS analysis module periodically updates the IP block lists and pre-checking policy based on the analysis of the violation logs. The pre-checked policy is then sent to the client-side SERFS.js by WASP runtime engine.

In a second experiment, we evaluated the performance impact with different size of SERFS templates. We initialized the LoadRunner with multiple users and increased the number of users gradually to send concurrent requests, and we ignored think time. We varied the size of SERFS templates as [0k, 30k, 60k, 90k] and plotted four curves for different sizes of the template, as shown Figures 16, 17 and 18. Figure 16 shows the plot for total number of transaction and Figure 17 shows the plot for average response time. We can see that the total number of transactions per second decreases by 19 points, and the response time varied from 442ms to 507ms when the SERFS templates size is increased from 0k to 90k. Figure 18 plots the cumulative completed transaction ratio against time. We can see that after 500 ms, the success transaction completed ratio decreases from 100% for 30k templates size to 68% for 60k template sizes.

5.2.2 SERFS Pre-checking

In order to improve the system performance when enabling friendly error message mechanism in server-side WASP we evaluated the performance improvement by dispatching SERFS pre-checking policy to the client browser. The WASP engine performs several different security actions to prevent the anomalous request as well as guarantee the user experience, including checking the request against vulnerability signature, generating friendly templates, dispatching SERFS.js components, and logging violation etc. We initialize 100 users in LoadRunner node and send concurrent requests to server side. For the 100 users, we simulate the malicious requests with attack characters and vary the ratio of malicious users as [0%, 10%, 50%, 70%]. We plot four curves for the average response time in Figure 19 with each represents a different attack ratio. As the number of suspicious users increase, the average response time increases as [444ms, 495ms, 517ms, 554ms]. This illustrates that the suspicious request will trigger more actions in WASP engine and cost more CPU utilization.

The SERFS pre-checking policy mechanism is designed to issues partial validation logic to do client-side pre-checking. If a pre-checking request still contains non-compliant characters, it will be identified as suspicious input

and be directly blocked by WASP. In Figure 20 and Figure 21, we compare the average response time and system transactions for ModSecurity and WASP with SERFS pre-checking mechanism.

Next we want to evaluate the impact of pre-checking as we increase the number of bad requests. Initially with no bad requests, there is a small pre-checking cost. As the ratio of bad requests to good requests is increased to 70%, the average response time of ModSecurity increases to 680 ms rapidly in Figure 20, and the transaction per second drops to 48 points in Figure 21. For SERFS pre-checking mechanism, the average response time is controlled within 500 ms, and the transactions per second is 69 points. Using pre-checking method, we can see that friendly responses are not needed for suspicious request, thereby improving the performance of the server-side WASP engine.

6 Related Work

The focus of this article is on providing smart and user friendly responses to clients whose requests have been filtered. In SERFS we do not modify any of the back-end application. To the best of our knowledge ours is the first work where we address one usability aspect of a WAF in some detail. There are several works on Web security that focus on different phases of software life cycle, including development [15], testing [2, 3], and runtime [7, 10, 13, 11] and analysis [8, 6] phases.

From the usability point of view, it is important to bridge the gap between a firewall and back-end applications [12]. There are few WAFs, such as ModSecurity, BroswerShiled, and F5 etc. [10, 13, 11] that adopt server-side application-level firewall to offer immediate assurance of security, which mainly have two challenges: one is the usability issue and the other one is performance issue. We found that when some non-compliant requests are detected these firewalls throw a static error page [11] to the end user, which breaks the user friendly behavior of application logic and provide bad user experience. Adaptive Security Dialog [4] and JSONR[7]provide user friendly programming model for the development of rules.

From the performance point of view, it is important to maintain the response time at acceptable level. Chong et al. describe a partition framework to distribute the security function among client browsers and servers automatically to balance security and performance requirement [1]. AJAXScope provides a mechanism to remotely monitor the client-side performance of JavaScript code [5]. Michiaki and Suzumura [14] use HTTP templates to separate the webpage presentation from its underlying business logic, and then off-load HTML generation tasks to Web clients to improve performance. In SERFS partial validation logic is issued to client side for the client-side pre-checking. If a pre-checked request still contains non-compliant requests, they will be identified as suspicious input and hence directly blocked without friendly services. By this way, the server side cost will be saved.

7 Conclusion

In this article we presented the design, implementation, and experience of user friendly feature of a Web application firewall. Using SERFS we showed how to enable user friendly response without modifying any of the back-end application, and with out revealing any of the sensitive security rules to end users. To improve performance we enabled client-side pre-checking of end user violation. The pre-checking also allows us to update security policy and use them to either block suspicious requests via IP address blocking or re-authenticate the session with the end user. In other words, if a pre-checked request still contains non-compliant requests, they will be identified as potential suspicious input and hence directly blocked without friendly message. We also presented several empirical results to illustrate capabilities of our approach that include both user experience and performance improvements with pre-checking features.

Acknowledgement

We thanks Ying Xin Pan for helping us with User Experience part of the experiments.

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

References

[1] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41:31–44, October 2007.

[2] 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.

[3] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[4] 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.

[5] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 17–30, New York, NY, USA, 2007. ACM.

[6] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[7] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 3:1–3:16, Berkeley, CA, USA, 2007. USENIX Association.

[8] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS07*, 2007.

[9] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM.

[10] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1, September 2007.

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

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

[13] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[14] Michiaki Tatsubori and Toyotaro Suzumura. Html templates that fly: a template engine approach to automated offloading from server to client. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 951–960, New York, NY, USA, 2009. ACM.

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