WSF: An HTTP-level Firewall for Hardening Web Servers

Xin Zhao and Atul Prakash
EECS Department, University of Michigan
1301 Beal Ave, Ann Arbor, MI, 48109
{zhaoxin, aprakash}@eecs.umich.edu

ABSTRACT

Due to both complexity of administration, insufficient checks on input data in many web applications, as well as lack of a single place to enforce security policy, web servers remain prone to external tampering. This paper proposes WSF (web server firewall) to protect web systems with three new mechanisms. First, WSF provides a language for specifying fine grained access control policy and enforcing it at the perimeter of a web server. Second, to prevent abuse of web application with malicious parameters, WSF allows web application developers to specify the restriction on application running parameters, rather than requiring them to enumerating all possible invalid input patterns, which substantially simplify input validation. Finally, WSF collects web user behavior statistics, which helps administrators to detect abnormal activities and adjust the access control policy heuristically.

KEYWORD:

Firewall, Attack Signature, User Behavior Audit

1. Introduction

Attacks against web systems represent a substantial portion of the total number of network intrusions. According to the 2002 DTI Information Security Breaches survey, 44% of surveyed companies had suffered web attacks in 2001[1].

To counter web attacks, most web servers enforce coarse-grained access control to restrict the execution of web applications within a specified directory that CGI programs must reside. One can also deploy intrusion detection systems or vulnerability assessment systems with known attack signatures to detect malicious requests and vulnerabilities.

Unfortunately, the above approaches leave a lot to be desired. Coarse grained access control mechanisms are not flexible enough and often leave loopholes to attackers. Most IDS systems and vulnerability assessment systems rely on known attack signatures to protect web systems. However, it is hard to keep the attack signature updated with respect to the large number of vulnerabilities discovered daily. Moreover, vulnerabilities may be introduced by custom web-based applications developed in-house. Many attacks are tailored to these applications and may not match any of known attack signatures. It is hard to enumerate all possible malicious request patterns.

This paper proposes WSF(web server firewall), an HTTP level firewall, as a supplement to existing solutions, to help combat web attacks. We first describe the threat model we address and then summarize the extent to which our approach can defend against web attacks.

Threat Model

Like network firewalls, WSF is primarily designed to handle external threats, rather than insider attacks on a web server. Unlike network firewalls, WSF is aware of HTTP protocol and is designed to prevent attacks only at that level. At present, WSF primarily focus on two categories of attacks:

1. ***Unauthorized accesses to sensitive files****:* Modern web systems usually provide coarse-grained access control to restrict that web applications can be invoked by web clients only if they reside in a specified directory (e.g., /cgi-bin). However, the coarse grained access control often gives attackers opportunities to exploit configuration error and compromise the web system. An example attack is what we will call the ***bypass execution*** attack. CGI programs that are invoked from user input by the web server often need to run helper scripts or programs internally. The intent of the programmer is that the helper programs should not be invoked directly by a client. For example, a CGI program may authenticate a user and then invoke a helper perl script to accesse a database if the user is valid. Unfortunately, if the helper program is put in the same directory as the CGI program, it can be invoked by a malicious client directly (via the web server, but without going through the parent CGI program). Thus, attackers can bypass the user authentication and violate web server security.
2. ***Abuse of CGI programs with parameters that violate the designed specifications*****:** CGI Developers are supposed to do input validation and filter out requests with invalid parameters, but they often fail to follow a sound security methodology and overlook the input error checking. Attackers can exploit the vulnerability of weak input validation to send CGI programs the parameters that do not meet the normal length or format restrictions and cause SQL injection or buffer overflow attacks[2]. For example, suppose that a CGI program uses the dynamically generated SQL command to create a new user account,

   INSERT INTO USER(name, id) VALUES($username, 100);

   Here, *$username* is a CGI parameter input by the user via a web form. The original purpose of this CGI is to create **only one** user account. However, if no input

validation applies, an attacker may input "*tom', 99), ('mary*" in the *$username* field, the user creation command is then generated as:

INSERT INTO USER(name, id) VALUES('tom', 99), ('mary',100)

Because many database systems, such as MySQL, allow users to insert multiple records in a line, this SQL command will allow the attacker to insert two records instead of one as expected. The reason of this SQL injection attack is a security bug: the user input validation is insufficient.

Level of Protection

WSF helps to protect against a wide-range of common vulnerabilities with the following three mechanisms:

1. To prevent unauthorized access to web files, WSF provides a language for specifying fine-grained access control policy and enforcing it at the perimeter of a web server. With this language, web administrators can classify web clients into variety of roles and specify their access permissions to web objects at the granularity ranged from directories to files. In addition, rather than allowing all files in */cgi-bin* directory to be executed by web clients, WSF allows a web application to be invoked only if it is explicitly specified as executable to web clients, which effectively prevents the bypass execution attack.
2. To thwart abuse of web applications, WSF proposes an input validity specification language to allow developers to specify the valid input patterns instead of requiring enumeration of all possible malicious inputs, which substantially simplifies the input validation task.
3. WSF also collects user behavior statistics on a per-user/per-IP basis. The behavior statistics can be used to detect abnormal web activities and heuristically change the access policy to proactively delay or block the requests from malicious users.

The rest of the paper is organized as follows. In Section 2, we describe related work. In Section 3, we illustrate the architecture and design of WSF. In Section 4, the implementation details are presented. In Section 5, we evaluate the WSF system. Finally, we make our conclusions.

2. Related work

Most web protection mechanisms fall in two primary categories: intrusion detection/prevention systems and vulnerability assessment systems.

Intrusion Detection/Prevention Systems

Most intrusion detection/prevention systems deployed to protect a website work at network level or application level.

Network based intrusion detection systems such as snort [3] can analyze network traffic to detect web intrusions. However, network-based intrusion detection is vulnerable to insertion and evasion attacks[4]. In addition, the network IDS needs to model how the application interprets the operations, but this is almost an impossible task without receiving feedback from the application.

Aiming at the problem of network based IDS systems, several application level IDS systems are proposed.

Mod_security[5] filters http requests that match specified attack signatures. However, it does not provide fine-grained access control, and is less effective in preventing unauthorized accesses like *bypass execution* problem. In addition, it is hard to keeping attack signatures updated and enumerating all possible malicious request patterns.

David Scott and Richard Sharp proposed the Security Gateway[6] to support CGI input validation based on application-level security policies, which is similar to WSF's input validity specification. The difference between WSF and Security Gateway is that WSF supports fine-grained access control and collects user behavior statistics that can be used to detect abnormal web behaviors and adjust the access policy heuristically.

WebSTAT [7, 8] detects intrusions against a web server by analyzing its logs. Like WSF, it also uses behavior statistics to infer abnormal activities. However, while WebSTAT allows an administrator to associate actions with the intermediate step of an attack, it is hard to stop one evil connection and avoid interrupting other valid connections at the same time, because WebSTAT is independent of a web server. For the same reason, WebSTAT does not prevent unauthorized access to web files. In contrast, WSF works as a module of Apache web server, it sit in line and stop malicious requests on site.

Vulnerability Assessment Systems

Various vulnerability scanners such as ISS Internet Scanner [9], Saint[10], LibWhisker[11], Nikto[12, 13] and Nessus[14], help assess a web system for loopholes before bad guys find them. They primarily rely on attack signature based checking, which makes them often raise false alarms or fail to detect critical vulnerabilities[15].

3. Design of WSF

3.1 System Overview

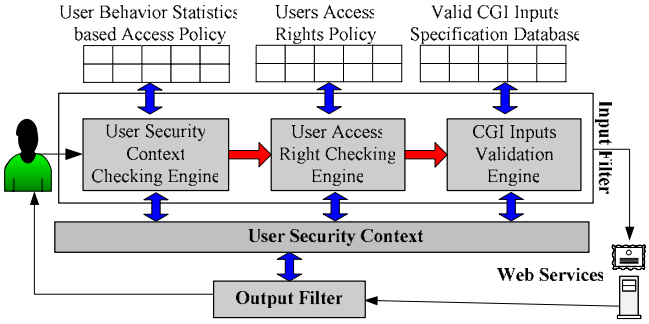

Figure 1. The architecture of WSF

As shown in Figure 1, WSF consists of the *input* and *output filters*. *Input filter* deep inspects the incoming HTTP requests to reject invalid web accesses. *Output filter* collects

the status of outgoing responses. Response status information helps infer user behavior patterns.

WSF maintains a per-user *security context.* A security context in WSF is indexed either by the user's IP address or by a user ID (if the user authenticated to the web service). We will defer the description on how to extract a user ID from web traffic to Section 4.2. The security context contains the user's past behavior statistics, such as the number of invalid requests, the number of failed requests, and the number of requests during a specified time interval. All those behavior statistics are updated by the input and output filters.

The input filter deploys three engines: *security context checking engine, access right checking engine*, and *CGI input validation engine*. These engines check the incoming requests one by one. An incoming request will be forwarded to the protected web server only if it goes through the checks of the three engines.

The *security-context checking engine* examines the user ID and the IP address of the request to see if requests from the IP address or the user ID should be blocked or delayed. Administrators can use the security-context checking engine to temporarily block a user's access to the web server if their statistical behavior, recorded in the security context, violates specified limits (e.g., too many failed authentication requests within a short interval). Therefore, the security context essentially works as a "credit history report" to help WSF monitor a client's abnormal behavior pattern and adjust its access policy accordingly.

The *access right checking engine* checks the requested URI against the access right policy. With the access right control, WSF can limit authenticated or unauthenticated users to only specified web files/services and prevent unauthorized access to the sensitive files that are left accidentally in public web directories. The access right checking engine provides fine-grained control, rather than standard access control imposed by web servers. Section 3.2 gives more details about the access right checking engine.

Finally, if the request is intended to invoke a CGI program, the request will be checked by the *CGI input validation engine*. The CGI input validation engine checks the parameters carried in the CGI request against the input validity specifications. Only requests with valid inputs can be sent to the web server. The CGI input validation helps mitigate many buffer overflow attacks and SQL injection attacks that compromise web systems via sending malicious parameters to CGI programs. More details are presented in Section 3.3

The *output filter* checks the status of outgoing replies and updates the behavior statistics in the security context. In addition, the output filter also helps the input filter to track the user information and generate the user tracking tag for each source.

3.2 Access Control Policy

WSF defines an access control policy language to allow administrators to explicitly define the access rights to web entries, including normal data files and CGI programs.

An access rule is a mapping as follows:

$$Web\_Entry \rightarrow Web\_User : Access\_Right$$

The *web entry* defines the object on which the access rule should apply. It can be a specific file, a class of files with a wildcard pathname or a directory. The *web user* defines the subject that is allowed to access the web entry. It can be a specific user or a web group. The *access right* defines the authorization under which a web user can access a web entry. The access right mapping means: the "*web_entry*" can and only can be accessed by the "*web_user*" under the "*access_right"* authorization.

An access policy usually includes three parts:

1. Definition of valid user set and user groups
2. Definition of default accessible file types
3. Definition of access right rules of web entries

The first part defines the valid user set and user groups.

The second part contains the default accessible file types (i.e. **.html* and **.jpg* files*)* for the web system. The accessible file types can be defined by file type extensions or certain file name patterns. By default, only common web file types are included, which helps prevent unauthorized accesses to sensitive files, such as "creditcard.dat", that are left in the public web directory.

The third part specifies the access right of users to web entries. An access right policy may include multiple access rules. Each rule defines the access right of one URI entry. A URI entry can be defined as a specific file, a class of files with a wildcard pathname or a directory. Wildcards are allowed and only allowed in file name to represent multiple files with similar name pattern. If an access rule defined for a directory, this access rule applies to all files and sub-directories under this directory if they are not associated with access rules. In other words, if no access rule is defined for a directory or a file, permissions are inherited from the parent directory. The access right rules are prioritized as follows:

$$root\ directory \rightarrow sub\text{-}directory(level1) \rightarrow$$
$$sub\text{-}directory(level2)... \rightarrow a\ class\ of\ files \rightarrow single\ file$$

The access rule of root directory has the lowest priority and access rules of single files have highest priority. Rules with higher priority have precedence in policy enforcement.

The CGI programs are treated differently. Each accessible CGI program must be explicitly specified to be executable. No wildcard is allowed in the access right rules for CGI programs. By default, only the CGI programs that are explicitly configured as executable can be requested to run by web clients. Thus, if a helper program, say "user_management.pl", is supposed to be only invoked by other trusted CGI programs, it will not be put in the access

right policy. Any attempts to directly invoke such a helper program via a URI will then be blocked by WSF.

3.3 CGI Input Validity Specification

Because the inputs to CGI programs are complex, fixed attack signatures are often not flexible enough to tell a valid input from invalid ones.

To deal with this problem, WSF provides a fine-grained way to specify constraints on inputs of CGI programs. We use an example to describe how validity specification works: suppose we have a user login script /cgi-bin/login.cgi, it only allows parameter transferred with POST method; the expected input at the user name field is a string composed by 3-8 letters or digits and the expected valid password is a string composed by 6-15 letters and digits. No special character is allowed in the username and password parameters. The validity specification can be defined as follows:

```
< Rule>
   <URI> /cgi-bin/login.cgi <\URI>
   < Method> POST <\ Method>
   < Parameter>
      <Name> username </Name>
      <Value> ^[a-zA-Z0-9]{3,8}$ </Value>
   </ Parameter>
   < Parameter>
      < Name> password </Name>
      < Value> ^[a-zA-Z0-9]{6,15}$ </Value>
   </ Parameter>
   <SIG_CHECKING> NO </SIG_CHECKING>
</Rule>
```

The *URI* section contains the URI of the CGI program.

The *Method* section configures which methods are allowed for this URI. The methods that are often used are GET and POST. Other HTTP methods like PUT, TRACK must be used carefully as they may bring vulnerabilities like cross site script attack[16].

The *Parameter* section defines the validity specifications for parameters of this CGI program. Each possible parameter must have a *Parameter* definition. The validity specification of each parameter consists of two parts: parameter name and parameter value. The parameter name field is the parameter name to be checked while the parameter value field shows the valid parameter value pattern. The valid parameter value pattern is defined with regular expression. If there is no restriction on a parameter, the valid parameter value pattern can be empty. Based on the configured validity pattern, the input validation checking engine can then check whether the user inputs carried in a CGI request is valid or not. Note that only parameters listed in this section will be regarded as valid and checked against the corresponding validity specification. For those parameters whose names are not on the valid parameter list, the input validation engine will directly regard them as malicious. This mechanism effectively prevents many buffer overflow attacks such as Code Red I and II attacks[17].

To reduce the risks of mis-configurations, the validity specifications can be tested with known attack signatures to see whether known attacks can slip through the protection of validity specifications. Currently, WSF use signatures extracted from the Snort attack signature database[3] to check the validity specification.

The above example shows, the rule clearly defines what inputs are expected by the programmer developers. The CGI program, at a minimum, must take care of inputs that satisfy the above specification. Any other unexpected inputs will be blocked by this specification directly at the firewall. This mechanism does not require developers to enumerate all possible invalid input patterns. Instead, web application developers only need to express their intention of valid inputs with regular express, which substantially simplify the input validation procedure.

3.4 User Behavior Auditing

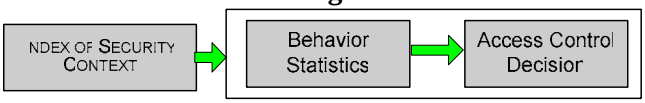

Figure 2. WSF Security Context

As a complementary mechanism, WSF also supports tracking and auditing of web user behaviors. WSF maintains a security context for each web client. The security context is indexed with the client's user ID if the client is an authenticated user. If the client is an anonymous guest, the security context is indexed with the client's IP address. As Figure 2 shows, the WSF security context contains three parts of user security information:

1. Index of the security context (User ID or IP address);
2. Behavior statistics;
3. Access control decision based on the behavior pattern.

WSF uses the *index of the security context*, IP address for unauthenticated user and User ID for an authenticated user, to locate a user's security context.

The *behavior statistics* part contains cumulative user behavior patterns, measured over multiple configurable time-intervals on a per-user/ IP basis:

- ***The number of received requests.*** This data is collected by the input filter.
- ***The number of bytes sent out.*** This data is collected by the output filter.
- ***The number of invalid requests***. This data is collected by the checking engines in the input filter. Any request that violates WSF security policies will be counted as an invalid request.
- ***The number of failed requests***. This data is collected by the output filter. Any request with the HTTP status code

that does not fall into the period between 200 and 307 will be counted as a failed request.

- ***The number of failed authentication requests.*** The field helps to prevent brutal force password guessing attacks. It is collected by the output filter.

The user behavior statistics help to detect abnormal behavior pattern and proactively adjust access control policies. For example, excessive authentication failures of a specific user may indicate that a hostile party is mounting brutal force password guessing attack or this user forgets the password. To thwart password guessing attack, web administrators can configure WSF to suspend this user's further authentication requests for several seconds upon the number of failed authentications exceeding the specified threshold.

4. Implementation Details

4.1 Modularized WSF

The Apache modularized architecture processes web traffic using the same idea as Unix command line filters: ps -ax | grep "apache.*httpd" | wc –l. The basic idea is to treat the information processing flow as an information stream. Apache modules can be inserted into the stream and organized as a module chain. Each module receives the data from upstream module, processes the data and then forwards the processed data to the next module in the chain. By this means, data in the stream can be manipulated independently from how it's generated.

With the same idea, WSF is implemented as an Apache module to terminate the incoming request, check it and decide whether to let the request go to next module. One advantage of deploying WSF as an Apache module is that the existing Apache code can be leveraged to reduce the implementation complexity. Another benefit is that WSF sits behind the SSL module and can monitor the decoded web traffic.

4.2 User Behavior Tracking

To collect a user's behavior statistics, WSF first needs to identify a web client. If the client is anonymous, WSF only needs to identify it by the client's source IP. If a client is an authenticated web user, WSF has to identify the user's ID to enforce the corresponding access policy.

To track the user identity, WSF requires the web administrator to fill out a login template to tell WSF the user ID field and successful authentication flag (i.e. a session cookie). With the login template, WSF's input and output filters cooperate with each other to track the user information. The input filter identifies the user authentication requests and extracts user information from the requests. With the extracted user information, the input filter generates a login memo to mark this request as an authentication request and save the extracted user information. The WSF output filter keeps checking whether an outgoing message carries the login memo. If it is, the output filter then searches for the successful authentication flags which are defined in the login template. If no success flag is found, the output filter regards the login request as failed. It simply forwards the outgoing message to the client and update the security context corresponding to the client's IP address. If the success flag is found in the response message, WSF infers that this is a successful authentication. The user associated with this authentication request becomes an authenticated user. WSF then generates a unique WSF cookie as the user identification tag. The WSF cookie will be carried with this user's further requests and used by the WSF system to track this user's activities. If no valid WSF cookie is located in an incoming HTTP request, WSF will always regard the request sender as an anonymous user.

5. System Evaluation

5.1 Security Evaluation

To evaluate the effectiveness of WSF system, we copied all files on our department website and deployed a parallel website as the testbed. Multiple attacks, including *Bypass execution*, *Random File Access*, and *SQL Injection*, are mounted against the testing website. The simulation results showed that WSF can effectively mitigate various web attacks.

5.2 Performance Evaluation

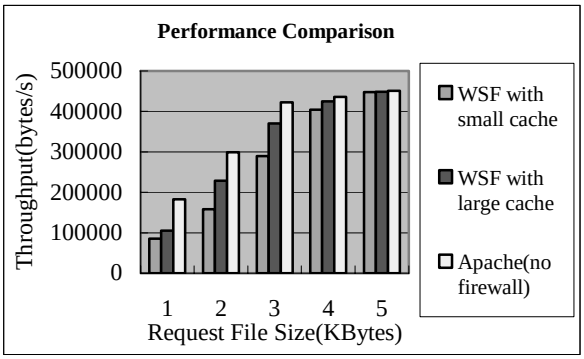

Figure 3 Throughput Comparisons

To evaluate the performance of WSF, we setup the simulation environment as follows: the web server is a Pentium IV PC with 1.8GHz CPU and 256MB memory with Linux 2.5.75 and Apache 2.0.40 installed. 3 Pentium III PCs with 850MHz CPU and 256MB memory work as web clients. Standard web system benchmark tools like WebStone does not support testing of authenticated web sessions that carry WSF cookies, we developed a benchmark tool that is similar to WebStone but supports authenticated web sessions. In the benchmark experiments, each of the three client hosts has 8 threads to send out HTTP request at their best efforts. Each thread sends 2000 HTTP requests in a sequential manner: a request will not be sent out until the reply of the previous request is received. In the simulation, we have deployed the access rules for 3394 web files and validity specifications for 150 CGI programs. The number of CGI validity specification rules has little effect on performance, because

the rules are indexed with CGI program pathnames and each CGI program is governed by one rule.

Figure 3 shows the throughput comparison of a web server with WSF support and without WSF support. We can see that when request file size is large, the apache server with WSF can achieve performance comparable to an apache server without WSF. However, when the requested file size is small, we can easily see performance penalties. The reason is that WSF is primarily CPU-bound. Most of its time is spent performing regular expression matching against client requests and updating behavior statistic records. When file size is large, the file transmission time is dominant, the WSF cost is relatively small. If file size is small, the CPU time used by WSF becomes non-negligible and thus reduces the apache server performance. However, as our prototype is completely un-optimized, we believe there is large scope to improve system performance. For example, Figure 3 also shows by increasing cache size to hold security contexts, WSF can achieve higher throughputs. This indicates that the size of memory allocated for caching security contexts can affect the system performance significantly. Upon receiving requests from a new client, the security context checking engine needs to load the client's security context from database into cache. If the cache is full, some clients' security contexts have to be sent back to the database. Those database I/O operations thus increase the system overhead. The larger the cache size is, the higher cache hitting rate is, and the less database accesses are required. Therefore, large cache helps to improve the performance of WSF.

6. Conclusion

WSF proposes a policy-based framework to provide perimeter security for those web services. With proper policies, WSF can help to thwart unauthorized accesses to system sensitive files and achieve flexible, role-based access control. To prevent attackers from sending maliciously manipulated requests to CGI programs, WSF allows administrators to explicitly define the input validity specification for each accessible CGI program. Instead of inferring all possible attacks from known attack signatures, WSF checks incoming requests against the input validity specification, which simplifies the procedure to determine whether a use input is valid or not. In addition, WSF collects user behavior statistics, which helps web administrators to detect abnormal user behaviors and proactively adjust the access control policies.

References:

1. BBC News, Web attacks on the rise, 2002. http://news.bbc.co.uk/1/hi/sci/tech/1930832.stm
2. Anley, C., Advanced SQL Injection In SQL Server Applications, 2002. http://www.nextgenss.com/papers/advanced_sql_injection.pdf
3. Roesch, M.S. *Lightweight Intrusion Detection for Networks*. in *Proc. of the USENIX LISA '99 Conference*. November 1999.
4. Ptacek, T.H. and T.N. Newsham., *Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection*. January 1998, Secure Networks.
5. Ristic, I., Introducing mod_security, 2003. http://www.onlamp.com/pub/a/apache/2003/11/26/mod_security.html
6. Scott, D. and R. Sharp. *Abstracting Application-Level Web Security*. in *Proceeding of the eleventh international conference on World Wide Web (WWW'2002)*. 2002.
7. Vigna, G., et al. *A Stateful Intrusion Detection System for World-Wide Web Servers*. in *Proceedings of the 19th Annual Computer Security Applications Conference*. 2003.
8. Kruegel, C. and G. Vigna, *Anomaly detection of web-based attacks* in *Proceedings of the 10th ACM conference on Computer and communications security* 2003 ACM Press: Washington D.C., USA p. 251-261
9. ISS, ISS Internet Scanner, 2004. http://www.iss.net/products_services/enterprise_protection/vulnerability_assessment/scanner_internet.php
10. SAINT Corp., SAINT vulnerability scanner. http://www.saintcorporation.com/products/saint_engine.html
11. rfp.labs, libwhisker. http://www.wiretrip.net/rfp/index.asp
12. Nikto, Nikto 1.32. http://www.cirt.net/code/nikto.shtml
13. Symantec Corp., Symantec NetRecon. http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=46
14. Nessus, NESSUS Scanner, 2004. http://www.nessus.org/
15. Forristal, J. and G. Shipley, Vulnerability Assessment Scanners, 2001. http://www.nwc.com/1201/1201f1b1.html
16. CERT Center, Microsoft Internet Information Server (IIS) vulnerable to cross-site scripting via HTTP TRACK method, 2004. http://www.kb.cert.org/vuls/id/288308.
17. CERT Advisory, "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, 2001. http://www.cert.org/advisories/CA-2001-19.html