WSF: An HTTP-level Firewall for Hardening Web Servers

Xin Zhao
EECS Department, University of Michigan
1301 Beal Ave, Ann Arbor, MI, 48109
zhaoxin@eecs.umich.edu

Atul Prakash
EECS Department, University of Michigan
1301 Beal Ave, Ann Arbor, MI, 48109
aprakash@eecs.umich.edu

ABSTRACT

Violations of confidentiality and tampering of data at web servers can cause significant costs and worry to corporations. A web system usually consists of multiple subsystems: web servers, data files, authentication services, and application programs that access database servers. Unfortunately, due to both complexity of administration, insufficient checks on input data in many CGI programs, as well as lack of a single place to enforce security policy, web servers remain prone to external tampering or leakage of confidential data. This paper proposes WSF (web server firewall) to help system administers provide perimeter security to their web systems via a flexible security policy framework. WSF provides a language for specifying security policy regarding permitted accesses and enforcing it at the perimeter of a web server. We show that, with proper policy specification, WSF can help mitigate many attacks such as buffer overflows via sending long parameters to CGI programs and SQL injection attacks. WSF access control model can be used to limit authenticated or unauthenticated users to only specified CGI programs/web services and with specified restrictions on parameters. User behavior statistics provided by WSF can also be tracked and compiled into per-user security contexts. This history can be used to heuristically change the access policy according to the user behavior patterns or to proactively block malicious users.

1. INTRODUCTION

Due to both complexity of administration and insufficient checks on input data in many CGI programs, web servers remain prone to external tampering or leakage of confidential data. According to the 2002 DTI Information Security Breaches survey, 44% of surveyed companies had suffered web attacks in 2001[18].

To counter web attacks, several schemes have been proposed. For example, URLScan [16] and mod_security [22], an Apache plug-in, reject requests that match specified attack signatures. Access control policy, on the other hand, is usually scattered around the system and is typically coarse-grained -- primarily managed via controlling access rights to directories in which CGI programs must reside. Usually, no administrative controls are provided to ensure that CGI programs are invoked correctly – that is usually left to programmers to filter out possible malicious parameters.

Unfortunately, the above approaches leave a lot to be desired from the perspective of consistent security policy control across all the web server components. An administrator rarely has a consistent view or control of the overall security policy. This paper describes the architecture of WSF (web server firewall), which helps system administers provide perimeter security to their web systems via a flexible security policy framework.

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 at that level, rather than at protocol levels below HTTP. We assume that attackers may attempt to carry out following types of attacks:

1. **Gain unauthorized accesses to executable files**: Unauthorized access to executable files often results from configuration errors [24]. An example vulnerability is what we will call the *bypass execution* problem. CGI programs that are invoked from a user input by the web server often invoke helper scripts or programs. The intent of the programmer is that the helper programs will not be invoked directly by a client. For example, the main CGI program may authenticate a user and then invoke a helper perl script that accesses a database with username as an argument. Unfortunately, if the helper program can be invoked by a malicious client directly (via the web server, but without going through the parent CGI program), it can bypass the user authentication and violate web server security.
2. **Take advantage of lack of fine-grained access control**: Programmers may have intended some CGI programs or data files to be used by only a subset of users. In addition, an administrator may want to restrict web clients to specified types of files. If only coarse-grained access control is used, all files in directory */html* are usually accessible by web clients. If some sensitive files like ".account.txt.swp" or "creditcard.bak" are left in that directory accidentally, a malicious user can access them directly from the web browser, which is apparently undesired. Fine-grained access control helps reduce this risk. While some existing web servers do support that, we would like a HTTP firewall to be useful in providing fine-grained access control.
3. **Invoke CGI programs with parameters that violate the designed specifications:** Programmers are supposed to check for such violations, but often overlook error checking. This security bug is often exploited by attackers to send parameters to CGI programs that do not meet the normal length or format restrictions and cause SQL injection or buffer overflow attacks[3]. 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 MySQL allows 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. We believe that an HTTP firewall should be an additional line of defense against violation of specifications since the firewall is under direct administrator's control.

Level of Protection

While we recognize that WSF is not a panacea, as an application firewall, it does help to protect against a wide-range of common vulnerabilities at the HTTP level:

- WSF provides a language for specifying security policy regarding permitted accesses and enforces it at the perimeter of a web server. WSF access control can be used to limit authenticated or unauthenticated users to only specified web files/services, which helps web system avoid the *bypass execution* problem, as well as inadvertent or malicious execution of unauthorized scripts in executable directories (even scripts that may be uploaded by an attacker).
- Instead of requiring an administrator to enumerate all possible malicious input patterns, WSF provides a language for specifying restrictions on parameters of web applications. With proper policy specification, WSF can help mitigate many attacks such as buffer overflows via sending long parameters to CGI programs and SQL injection with unknown input patterns.
- User behavior statistics can be tracked and compiled into per-user security contexts. This history can be used to heuristically change the access policy according to the user behavior patterns to proactively block malicious users.

Note that IP spoofing attack is not addressed by WSF. The source IP is of course easy to spoof. However, since HTTP-based web systems are connection oriented, hijacking a HTTP connection with a spoofed IP address is usually much harder and can be detected from abnormal network traffic by network firewalls [12]. We do not consider this attack further in this paper.

Intruders can also subvert a website by compromising the web server or other system files like */etc/passwd*. However, as an application firewall, WSF does not prevent system-level attacks. Those attacks can be mitigated by system level security solutions such as SELinux and LIDS[14, 29].

We also note that WSF can filter incoming HTTP requests at runtime but, at present, it does not scan for virus or malicious scripts carried in the files that are uploaded to web servers. Mod_security has this feature. We believe such a capability could be added in future versions of WSF without significant impact on its design or it could be handled by a separate IDS system or a virus scanner. WSF will, however, prevent execution of malicious scripts, as pointed out earlier, unless the attacker is able to replace existing scripts by malicious ones that have identical names and take same parameters with same formatting restrictions. Use of cryptographic hashes on scripts could help eliminate this possibility, but our current version does not support that feature at present.

Finally, we point out that even though WSF can catch many errors that may be overlooked in CGI programming, it does not eliminate the need for safe CGI programming or other good security practices. We consider WSF to be complementary to them. WSF should be used as the first line of defense in web server security, rather than the only form of security.

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

In this section, we describe several related systems that can potentially protect websites. We also briefly analyze the difference between these schemes and WSF. 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 [23] can analyze network traffic to detect web intrusions. However, network-based intrusion detection is vulnerable to insertion and evasion attacks[20]. 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. Minor differences in operations play a major role in how they are interpreted. For example, request *http://www.someHost.com/dir1\file1* (note the backslash character) sent to Microsoft-IIS/5.0 will make the web server to lookup file *"file1"* in the directory *"dir1"*. However, if the server is Apache version 1.3.6, the server looks for a file named *"dir1\file1"* in the root directory. In addition, only a few IDS systems use information about web applications and detect malicious sources with user behavior statistics. Encrypted web communication imposes another challenge to network based IDS systems. To understand encrypted web communications, administrators might have to deploy another web proxy system for an IDS system, which further increase the maintenance cost and may reduce system performance.

Aiming at the problem of network based IDS systems, paper [2] proposed to collect data real-time transaction information from web server and forward it to an IDS system. However, it is only a data collection system instead of a web protection system.

Mod_security[22] is an application firewall deployed in Apache server that can not only collect data but also analyze incoming requests and prevent intrusions onsite. However, mod_security does not prevent unauthorized accesses like bypass execution problem or execution of old scripts accidentally left in an authorized directory while WSF does. In addition, mod_security relies on attack signatures to detect malicious inputs, which is less effective in preventing malicious inputs with unknown patterns.

David Scott and Richard Sharp proposed the Security Gateway[25] to support CGI input validation based on application-level security policies. The difference between WSF and Security Gateway is that WSF identifies web clients and collects user behavior statistics that can be used to adjust the access policy heuristically. Furthermore, Security Gateway does not support access control and thus cannot prevent attacks such as bypass execution.

WebSTAT [13, 28] 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 but avoiding interrupting other valid connections, because WebSTAT is independent of a web server. In addition, WebSTAT detects

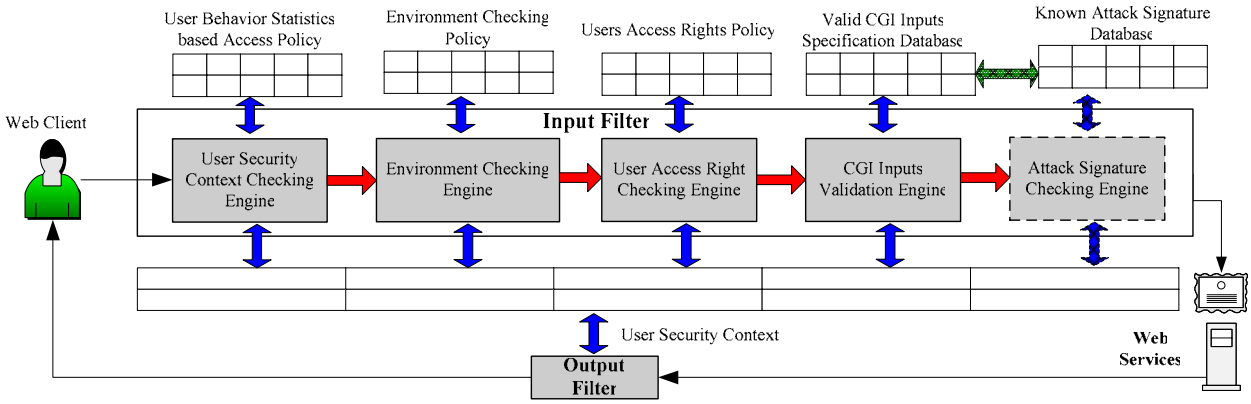

Figure 1. The architecture of WSF

malicious inputs with malicious input patterns, which requires WebSTAT enumerate all possible malicious input patterns. In reality, the enumeration of malicious input patterns is a tough job. In contrast, WSF allows an administrator to customize for each web application what input patterns are valid. The validity specification is therefore more effective in preventing malicious inputs with unknown patterns assessment systems.

Vulnerability Assessment Systems

Various vulnerability scanners such as ISS Internet Scanner [11], Saint[7], NetRecon[8], Retina[26], CyberCop Scanner[27], LibWhisker[21], Nikto[8, 19] and Nessus[17], can help assess a web system for many loopholes before bad guys find them. However, because of lack of detailed information of web applications, they can only do signature-based checking. However, that makes them often raise false alarms or fail to detect critical vulnerabilities[10]. In [10], Forristal and Shipley set up 17 of the most common and critical vulnerabilities for those scanners to check. The closest of 6 tested famous scanners was the Nessus Security Scanner, which failed to detect only 2 of the 17 common vulnerabilities[10]. To assess the current state of these scanners, we intentionally put the SQL injection vulnerability described in Section 1 into our site and used several scanners including Nessus to check for it. With their standard signature lists, all scanners we tried failed to find this loophole.

Scanners are severely limited in detecting unauthorized access to files or scripts, because they have no information about appropriate access rights. For example, no scanners we tested can detect the *bypass execution* vulnerability described in Section 1 because they cannot distinguish between helper programs and real CGI programs.

3. DESIGN OF WSF

In this section, we present the architecture and design of WSF. WSF provides three capabilities to assist the protection on web systems. First, WSF defines a language for specifying policy regarding permitted accesses and enforces the policy at the perimeter of a web server. Second, WSF provides a language for specifying restrictions on parameters of web applications, which helps mitigate the attacks that subvert web applications via sending maliciously crafted parameters. Finally, WSF can track user behaviors and compile behavior statistics into per-user security contexts, which can be used to adjust security policies to proactively block malicious users.

3.1 System Overview

As shown in Figure 1, WSF consists of the input and output filters. Input filter deep inspects the incoming HTTP requests to detect and prevent 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 interval. All those behavior statistics are updated by the input and output filters.

The input filter deploys five engines: *security context checking engine, environment checking engine, access right checking engine, CGI input validation engine* and *Attack Signature Checking Engine*. These five 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 five engines.

When a new HTTP request is received, the request is first processed by the *security-context checking engine*. This engine only 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. 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 requests within a specified 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 to prevent denial-of-service style attacks at the HTTP protocol level.

If the security context shows that the website is currently accessible for this client, the request is next processed by the *environment checking engine*. Some web attacks are due to malicious request environment variables like HTTP headers. For example, the PHPNuke Admin Cookie Variable SQL Injection attack is launched by putting malicious cookies in HTTP headers[9]. The environment checking engine helps to filter out requests with invalid environment variables.

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 directory. The access right checking engine provides substantially more fine-grain control than that in Apache or mod_security. Section 3.2 gives more details about the access right checking engine.

If the request is intended for a CGI application, 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. This capability of validating CGI inputs in WSF goes substantially beyond the filtering capability in mod_security. We will give more details in Section 3.3

Finally, the *attack signature checking engine* checks the request against known attack signature database. By searching attack patterns in the request, the attack signature checking engine helps to filter out malicious requests that slip through the checking of the first four checking engines. The checking of the attack signature checking engine is optional while other four engine's checking is mandatory.

The output filter checks the status of outgoing reply 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.

The rest of this section presents the three capabilities WSF provides, respectively.

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.

The basic principles we follow are the following:

a) *Fine-grain access conrol:* Access rights to files/scripts can be controlled on a user/group basis. A user can be identified by its IP address or an authenticated user identity, and thus can be used to permit or deny access to sections of the web site on basis of both source IP address as well as web identity of authenticated user.

b) *Precedence Rule:* More specific access right specifications have precedence; For example, if the access rights of /www/html/ and /www/html/subdir1 are different, and there is a request for /www/html/subdir1/foo.html, the access right on /www/html/subdir1, rather than /www/html/, will be enforced for this request.

c) *Explicit user to executable mapping:* Scripts or executable programs (defined by suffixes such as .cgi, and .sh) must be explicitly specified to be trusted and executable in the WSF policy for them to be invoked by a web request. We follow the principle in the Clark-Wilson integrity model that an explicit mapping must be defined between users and trusted procedures[6].

An access rule is essentially 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. Like a normal operating system, a WSF administrator can setup a set of web users and groups. Each WSF user has a user ID that can be mapped to the user ID of a real web service. All user information (ID and user mapping) is put in a file called "*user_account_file*". Each group can contain arbitrary web users whose IDs can be kept in the group member file.

The *access right* defines the authorization under which a web user can access a web entry. Currently, possible access rights can be *"accessible"* or *"executable"*. But more access rights can be added in the future.

Therefore, the access right mapping essentially 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. It can be defined as follows:

```
/*****1: USER&GROUP DEFINITION ******/
<Register_User>
    user_account_file    /* file used to store web user Identities*/
</ Register_User >
<Group>
  //"cs_group" is the group name
  <Group_Name> cs_group </Group_Name>
  < Group_Member >
     cs_group_file    /* file used to store group member IDs */
  </ Group_Member>
</ Group >
```

The names of valid users are put in an account file *"user_account_file"* and loaded with WSF module. Likewise, the user identities of group members are put into the corresponding group files. The group name "public" is reserved for the group that consists of all web users, including anonymous users.

The second part contains the default accessible file types in the web system. It can be defined as follows:

```
<Accessible_File_Type >
    *.html *.htm *.gif *.jpg *.pdf
</Accessible_File_Type>
```

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 the sensitive files that are left in the public web directory. Sensitive files like "creditcard.dat" are inaccessible even if they are in the public directory because of disallowed suffix. While this mechanism does not completely prevent access to all sensitive files, it can significantly reduce the risk of unauthorized accesses to sensitive files.

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 → sub-directory(level1) → sub-directory(level2)… → a class of files → 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.

A simple example can illustrate the access right rule format:

```
/* this rule specifies that only admin user and faculty members are
   allowed to access web directory /coursetool */
<WWW_Access_Rule>
    <URI> /coursetool </URI>
    <ACCESS_RIGHT>
        <User> admin </User>
        <Group> faculty </Group>
        <Right> Accessible</Right>
    </ACCESS_RIGHT>
</ WWW_Access_Rule >
```

By parsing the access right policy file, the access rules are mapped to a directory tree structure. The root of the tree is the web root directory; each intermediate node represents a subdirectory under the web root directory; the leaf nodes represent web files or subdirectories. Each node is associated with its access right. When retrieving the access right of a request URI, the access right checking engine always goes down the tree to find the node that has closest match with the requested URI. The access right is then retrieved from the matching node to apply on the request URI.

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. For example, to prevent SQL injection attacks, the stored procedure name "*xp_cmdshell*" is often regarded as an invalid input pattern[3]. However, if the web service is developed to help users to check the syntax of SQL language, "*xp_cmdshell*" is a valid input for users to use this service. Under this situation, fixed signature-based checking may regard the "xp_cmdshell" query as invalid and raise false alarms. Another example, as described in Section 1, inputs like "***tom', 99), ('mary***" are often invalid because they may trigger SQL injection attack. However, this input pattern may be valid for applications irrelevant to SQL-based operations. Putting a fixed input pattern as an attack signature may prevent normal use of web applications.

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>
```

Each validity specification rule is bracketed with the <Rule> and </Rule> pair and consists of three sections: *URI*, *Method*, *Parameter* sections and an optional *SIG_CHECKING* section:

The *URI* section contains the URI of the CGI program. Only if the requested URI matches the defined URI section, will the input validation engine checks the rest of the rule. If a requested URI does not have a matching rule, the request will be blocked directly.

The *Method* section configures which methods are allowed for this URI. According to the HTTP protocol specification, besides GET and POST, other methods like PUT, TRACK are also supported by many web systems, which can bring vulnerabilities like cross site script attack[5]. To prevent unnecessary method uses, web application developers can explicitly define which methods are allowed. Usually, for sensitive information transfer, only POST method should be allowed. But developers can configure additional methods separated with colon.

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[1].

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[4] to check the validity specification. Attack signatures that can go through the checking of validity specification often imply that the validity specification may be too loose. The policy maker will then be prompted for verification. Offline checking helps to improve system performance. However, if it is difficult for the administrator to define a parameter rule to express his demands and exclude the attack signatures, he can rely on the signature checking by setting the *SIG_CHECKING* field to true to make the parameter checked against the attack signature database.

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 validity specification does not prevent hidden value manipulation attacks in CGI programs. While dynamically generating a MAC for hidden values can easily prevent the hidden value manipulation attack, it significantly reduces system performance[25]. On the other hand, a static analysis tool can easily find which file uses hidden values and warn administrators to avoid the use of hidden fields.

3.4 User Behavior Auditing

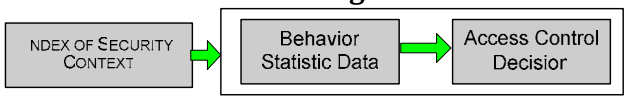

Figure 2. WSF Security Context

The third capability WSF provides is the user behavior tracking and auditing. In a WSF system, each client has a security context. 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:

- Index of the security context (User ID or IP address);
- Behavior statistics;
- 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 statistic data* part contains cumulative user behavior patterns, measured over multiple configurable time-intervals on a per-user 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.

Based on the user behavior statistics, an administrator can specify dynamic access control policies according to user behavior pattern. For example, a large amount of failed authentication requests from a single client may be because this client is mounting brutal force password guessing attack. The administrator can put a policy to suspend this client for a period of time if a user has excessive failed authentication attempts.

Following are examples of malicious behaviors that the user behavior statistics can help detect:

- Probing of hidden files at a web server by generating possible file names and trying them out exhaustively. This attack is indicated by return of many HTTP 404 error codes, which will be considered failed requests.
- Dictionary attack on user id/password. This attack is indicated by return of many authentication failures.
- DoS attack by sending a large number of requests in quick succession over an interval from the same IP address. This attack is indicated by monitoring lots of requests within short intervals, including valid ones.
- Stealing of data in a database or overloading a server. Unusually, large number of bytes fetched by the same user or IP address within short time is an indicator of this attack.

4. IMPLEMENTATION DETAILS

The prototype of WSF is developed for Apache server. But the idea of WSF can be adapted to support other web servers. In this section, we first describe how WSF fits into an Apache server to protect the web system. Next, we illustrate how WSF identifies a web client and tracks this user's behavior.

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, it usually has lowest privilege in web entry accessing. WSF only needs to identify it by the client's source IP. This client's behavior statistics can be collected on the basis of source IP.

On the other hand, if a client is an authenticated web user, it may have higher privileges in accessing web files and executing CGI programs than an anonymous user. WSF has to identify the user's ID to enforce the corresponding access policy. However, a user's identity is usually authenticated and maintained by CGI programs. How to extract the user's ID and trace its following behavior is a challenge. Because a client must go through the authentication process to become an authenticated user, WSF can extract user ID by analyzing the user authentication process.

Existing web services usually do user authentication with one of three major approaches -- user credential, IP source filtering, and authenticated proxy(only access request from authenticated proxies and the proxies authenticate end users)[15]. According to different authentication methods, WSF uses different ways to extract user information from the authentication communications. Due to the limit of space, this paper only describes how to extract user ID if credential-based user authentication scheme is used.

In credential-based user authentication scheme, either via a web form or methods embedded in HTTP protocol, the user presents some form of credential -- user ID and password, or a cryptographic certificate -- to the web service provider as evidence that he or she is a legitimate member of the user community. The web server then validates this credential using some user authentication programs. A successful authentication typically results in a special cookie that is returned to the client as the user's identity for future communication. WSF extracts the user information just from this user authentication process.

To track the user identity, WSF requires the web administrator to fill out a login template to tell WSF how to track the user identity. The login template is as follows:

```
<USER_LOGIN_TEMPLATE>
    <URI> $CGI_URI </URI>
    <Parameter>
        < ParameterName> $NAME </ ParameterName>
        < ParameterValue> $VALUE </ ParameterValue>
                    ……
        < ParameterName> $NAME </ ParameterName>
        < ParameterValue> $VALUE </ ParameterValue>
    </Parameter>
    <Name>$USERNAME</Name>
    < Success_Flag>
        <Cookie> $COOKIENAME </ Cookie>
        <Flag_String> $Success_String </Flag_String>
    </ Success_Flag>
</USER_ LOGIN _TEMPLATE>
```

This template tells WSF how the web system authenticates users:

- The *URI* field shows what CGI program authenticates users. Some websites integrate all user account related services into a single CGI program; different services are differentiated with different parameters such as service ID. To avoid confusion, WSF also supports using the URI field combined with the *Parameter* fields to identify a login service. Hence, *Parameter* fields are optional.
- The *USERNAME* field tells WSF the name of the parameter that is used to store the user name related to this authentication process. With this field, WSF can extract the user ID from an authentication request and track the activity of this user.
- The *Success_Flag* field shows the flags of a successful user authentication. A successful user authentication usually results in a cookie or a webpage with some specific strings. WSF can search for these flags to determine whether the user creation is successful or not. The *Success_String* can be defined using regular expresses.

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 indicate this request is an authentication request. In addition, the extracted user information is also saved in the login memo. This login memo is associated with the login request and will be carried by Apache throughout the lifetime of this login request.

The WSF output filter, as another block in the Apache information stream, 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

In this section, we first evaluate the effectiveness of WSF system. Next, we give the scalability evaluation. Finally, we present preliminary performance evaluation results.

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. We mounted the following attacks against the simulated website:

1. *Bypass execution*: By searching the files on the website, we did find some helper perl scripts in the /cgi-bin directory. They are supposed to be called by other CGI programs only. However, without WSF system, they can be invoked from a client's web browser. By explicitly specifying the executable files, WSF effectively prevents the bypass execution attack.
2. *Random File Access*: In the /html directory, we noticed that there existed some swap/backup file like *".intro.html.swp"* and *"foo.bak"*. Those files can be read from a web browser due to the coarse grained access control. With WSF, we can prevent this attack by specifying the types of accessible files. By excluding ".bak" and ".swp" files from accessible file types, WSF can block requests to those files. Furthermore, if we keep trying to access various potential files, we are suspended by WSF because WSF detects from the security context that we sent excessive failed HTTP requests.
3. *SQL Injection*: In the student resume service, one of our departmental web services, we detected a loophole that allowed us to mount the SQL injection attack described in

Section 1. With WSF, we specified the valid specification on user inputs, which effectively blocked malicious inputs

4. *Buffer Overflow*: We did not find a buffer overflow vulnerability on the simulated website. Instead, we intentionally put an application that vulnerable to buffer overflow attacks on the website. By exploiting this vulnerable application, we successfully gained the access of user "apache". On the other hand, by specifying input valid specification, WSF caught the malicious request.

These simulated attacks showed that WSF can effectively mitigate various web attacks. However, we also noticed that if mis-configured, WSF also let certain attacks slip through. For example, in the attack test of random file access, if WSF specified that "*.*" as accessible type, attackers can still access any files accidentally left in directory /html. Therefore, WSF does provide mechanisms to mitigate web attacks, but it is not a fool-proof solution and still needs right configuration.

The configuration of the WSF protected website does incur some deployment cost, but most of it is one-time cost since web entries are not likely changed frequently. If security is paramount, we believe that the deployment and incremental cost of using WSF is reasonable, especially given the benefit of long-term control over preventing sophisticated attacks against a website.

5.2 Scalability Evaluation

One scalability concern is about the memory size used to cache security contexts. WSF uses a two-level storage to keep the security contexts: the most frequently visited security contexts are kept in an in-memory cache while other security contexts are stored in MySQL database. If a client's security context is not in the cache, WSF will load it from the database. With the number of visiting clients increases, the overhead of maintenance on security contexts may also increase. However, we believe that the memory size required to cache security contexts is acceptable: assuming a website is hit for 1 million (10^6) times a day and most of the hits occurs during a 10 hours peak time period. If each visiting client sends out 5 requests at average before it leaves, there will be $2*10^5$ visiting users in total. Because each security context takes less than 50 bytes of memory, 10M (10^7) bytes memory can cache more than $2*10^5$ users and is enough to hold the security contexts for all visiting clients, which is quite manageable.

Another scalability concern is about the memory requirement for caching access right policies and CGI input validity specifications. Assuming there are 10^5 web files in the system, under the worst condition, the administrator does not use wildcard to define access rights. Each file needs one separate access rule. Since each access right entry usually takes less than 50 bytes, the total memory used to cache the access right is less 5M, which is not a big concern for modern computers. The overhead access right retrieval is trivial since the searching process is essentially a tree traverse process.

In addition, to maintain a client's security context, WSF needs at most two database accesses (load the security context from database and write it back to database when it is swapped out) per day, assuming a user's security context will be cached for one day if memory allows. The database access cost becomes another scalability concern. In the scenario described in the above theoretic analysis, the total database accesses during 10 hours will be at most $4*10^5$ ($2*2*10^5$) times. The average database access rate will be less than 11.11 times/second. Because modern databases can handle more than 150 primary key indexed queries per second[4], we believe that the cost of database access is quite manageable. The security contexts in memory are hash-indexed, the cost of access in cache is trivial.

5.3 Performance Evaluation

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. Each client 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. In addition, we defined 8 environment checking rules.

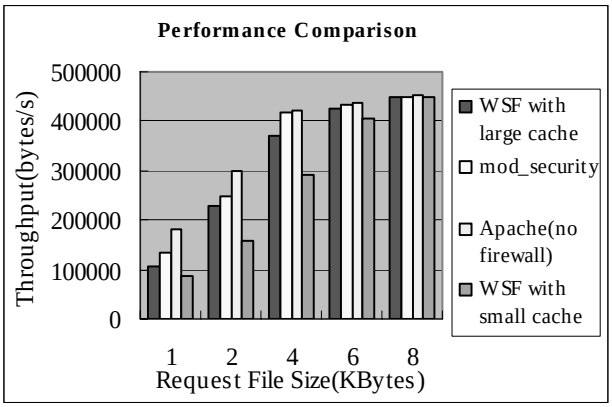

Figure 3 Throughput Comparisons

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 prevent 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 to the access control and CGI parameters checking, WSF also tracks user activities and supports heuristically changes of the access policy according to the collected behavior statistics, which helps WSF to detect abnormal user behaviors and proactively block malicious web clients.

Bibliography

1. CERT Advisory, "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, 2001. http://www.cert.org/advisories/CA-2001-19.html
2. Almgren, M. and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. in *RAID '00: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*. 2001: Springer-Verlag. p.22--36
3. Anley, C., Advanced SQL Injection In SQL Server Applications, 2002. http://www.nextgenss.com/papers/advanced_sql_injection.pdf
4. Carr, B., MiniSQL vs. MySQL: Performance, 1999. http://www.os2ezine.com/v4n2/p-sqlperf.htm
5. Center, C., Microsoft Internet Information Server (IIS) vulnerable to cross-site scripting via HTTP TRACK method, 2004. http://www.kb.cert.org/vuls/id/288308
6. Clark, D. D. and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. in *IEEE Symposium of Security and Privacy*. 1987. p.184-194
7. SAINT Corp, SAINT vulnerability scanner. http://www.saintcorporation.com/products/saint_engine.html
8. Symantec Corp., Symantec NetRecon. http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=46
9. Security Focus, PHPNuke Admin Cookie Variable SQL Injection Vulnerabiliy. http://www.securityfocus.com/bid/6890
10. Forristal, J. and G. Shipley, Vulnerability Assessment Scanners, 2001. http://www.nwc.com/1201/1201f1b1.html
11. ISS, ISS Internet Scanner, 2004. http://www.iss.net/products_services/enterprise_protection/vulnerability_assessment/scanner_internet.php
12. Joncheray, L. A Simple Active Attack Against TCP. in *Proceedings of 5th USENIX UNIX Security Symp*. 1995. p.7-19
13. 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
14. Loscocco, P. and S. Smalley. Integrating flexible support for security policies into the Linux operating system. in *Proceedings of FREENIX 2001*. 2001.
15. Lynch, C., A White Paper on Authentication and Access Management Issues in Cross-organizational Use of Networked Information Resources, 1998. http://www.cni.org/projects/authentication/authentication-wp.html
16. Microsoft, UrlScan Security Tool. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/tools/urlscan.asp
17. Nessus, NESSUS Scanner, 2004. http://www.nessus.org/
18. BBC News, Web attacks on the rise, 2002. http://news.bbc.co.uk/1/hi/sci/tech/1930832.stm
19. Nikto, Nikto 1.32. http://www.cirt.net/code/nikto.shtml
20. Ptacek, T. H. and T. N. Newsham., Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. January 1998, Secure Networks.
21. rfp.labs, libwhisker. http://www.wiretrip.net/rfp/index.asp
22. Ristic, I., Introducing mod_security, 2003. http://www.onlamp.com/pub/a/apache/2003/11/26/mod_security.html
23. Roesch, M. S. Lightweight Intrusion Detection for Networks. in *Proc. of the USENIX LISA '99 Conference*. November 1999. p.229-238
24. Rubin, A. D., D. Geer, and M. Ranum, Web Security Sourcebook. 1997: John Wiley & Sons, Inc.
25. Scott, D. and R. Sharp. Abstracting Application-Level Web Security. in *Proceeding of the eleventh international conference on World Wide Web (WWW'2002)*. 2002. p.396-407 http://www2002.org/CDROM/refereed/48/
26. eEye Digital Security, Retina Network Security Scanner. http://www.eeye.com/html/Products/Retina/
27. McAfee Security, CyberCop ASaP: Online Vulnerability Assessment Service from SecureSynergy. http://www.securesynergyonline.com/content/cybercop_asap/default.asp
28. 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. p.34-43
29. Xie, H., LIDS: Linux Intrusion Detection System. www.lids.org