Intercepting Filter Approach to Injection Flaws

Ahmed Salem*

Abstract—The growing number of web applications in the global economy has made it critically important to develop secure and reliable software to support the economy's increasing dependence on web-based systems. We propose an intercepting filter approach to mitigate the risk of injection flaw exploitation- one of the most dangerous methods of attacking web applications. The proposed approach can be implemented in Java or .NET environments following the intercepting filter design pattern. This paper provides examples to illustrate the proposed approach.

Keywords—Injection Flaws, SQL Injection, Intercepting Filter, Cross-site Scripting Vulnerability

1. INTRODUCTION

Over the course of its relatively short life, the Internet has evolved from drab collections of publicly available documents into an essential business tool that provides interactive contents and services. A large majority of the interactive contents and services delivered through the Internet are provided by websites that serve dynamic pages. These pages often contain context-sensitive information. Websites that are made of dynamic pages are referred to as web applications. These applications can provide most of the power and flexibility of a traditional application, and are typically implemented using the same languages and technologies.

Throughout the evolution of the Internet, the global economy has become increasingly dependent on web applications to provide services and conduct business. The practice of utilizing web applications to conduct business is known as Electronic Commerce or eCommerce, which has become one of the most important aspects of the Internet. ECommerce allows people and businesses to exchange goods and services immediately, without the barriers encountered with traditional storefronts. ECommerce is assisted by the existence of the Search Engine. The search engine provides a service that makes all the resources on the net available to search [13]. Apart from their contribution to the ecommerce venture, web applications have also crept into leading roles in the communication and defense tools for the United States and many other nations.

Coinciding with their ever-increasing importance, web applications have become an increasing target of malicious attacks. Most web applications use a database to store their data. The data thus stored, most often contains confidential or even sensitive information that should not be made available to the public [9]. Due to the "highly available" nature of web applications, they provide a large surface area for potential vulnerabilities, such as: SQL (Structured Query Language) injection and Cross-site scripting. These vulnerabilities are often exploited and used to attack web applications. Perhaps the most critical of these web application vulnerabilities are

Manuscript received November 2, 2010; accepted November 22, 2010.
Corresponding Author: Ahmed Salem
* Department of computer science at California State University, Sacramento(CSUS), USA (salema@ecs.csus.edu)

injection flaw exploitations, which are frequently used to attack back-end databases and other software systems.

An Injection flaw is a flaw in the web application interface which allows attackers to relay malicious code through those interfaces to other systems. These attacks can include calls to the operating system via system calls, uses of external programs via shell commands, or calls to back-end databases via SQL (Structured Query Language), leading to SQL injection. SQL injection opens the database so the hacker can obtain unlimited access to the now unsecured data in the database. The easiest way to find an SQL injection is to insert some invalid characters and observe what error messages the server responds to the browser with [14]. This can be dangerous because the error message can display the full directory in which the file is located. Malicious scripts written in Perl, Python or other scripting languages, can be injected into poorly designed web applications and executed. Moreover, some of the well known websites are also becoming victims of this injection, such as: Travelocity, Creditcards.com and Tower Records [10]. Any time a web application uses an interpreter of any type; there is the danger of an injection attack.

Many current websites process their content dynamically by using any of the available server side scripting each time a user requests a web page. This process allows a hacker to use malicious script to trick and attack the server. If the hacker discovers that the server does not validate, then he can use Cross-site scripting (XSS) attacks to exploit the server. Cross-site scripting is abbreviated as XSS since CSS (Cascading Style Sheets) is already widely known as Cascading Style Sheets. XSS gained its popularity in 2001. XSS works by taking an input from one user and outputting it to another user. The hackers normally will create a script that prompts a message to the user to type his password again, and then steals it [15]. As an example, if an attacker finds XSS vulnerability in a web page that processes credit cards, then the attacker can infuse malicious code into the page, where it would sit inside waiting for a victim. When someone tries to process a credit card using the page with the malicious code inside, it would send all the vital data to the hacker without the user's knowledge.

The prevalence of injection vulnerabilities within the internet is enormous. Vulnerable websites with the injection flaws are being attacked and damaged every day. The latest significant example is an attack on the United Nations website on August 12, 2007 [16].

This paper outlines an intercepting filter approach aimed at increasing the security and reliability of web applications by eliminating injection flaw exploitations. We have also focused on preventing Cross-site scripting (XSS), which is one of the commonly occurring vulnerabilities in injection flaw exploitations. The remainder of this paper is organized as follows; section 2 discusses the related work, section 3 describes the intercepting filter approach, section 4 addresses the approach’s benefits and limitations and section 5 provides the conclusion and explains future work.

2. RELATED WORK

Injection flaw exploitations are widely recognized security problems within the domain of web applications. As such, there are a number of approaches and tools available with the goal of mitigating the risk of injection flaw exploitations. The Open Web Application Security Project (OWASP) provides a detailed approach for protecting web applications from Injection Flaws [1].

The OWASP approach recommends that web applications avoid access to external interpreters such as operating system shell interpreters, whenever possible. Avoiding these external interpreters reduces a large number of problems related to shell command injection. For the external interpreter calls that can not be avoided, such as calls to the back-end of a database, Halfond and Orso proposed a technique that uses a program to automatically build a model of the legitimate queries that could be generated by the application [2]. In addition, SPI Dynamics also suggest using regular expressions for sanitizing data before it is executed by a back-end database [3]. The use of regular expressions can mitigate the threat of SQL injection flaws by sanitizing data of potentially malicious content. This can be enforced with another filter on the server that verifies that no illegal characters are used. In addition to avoiding external interpreters, web applications can be secured by using static analysis and runtime protection [4].

OWASP recommends that a thorough validation of any inputted data needs to be made in order to ensure that the data does not contain any malicious content; this is something that can be implemented by a filter. In addition to avoiding external interpreters, OWASP recommends structuring requests in a manner that ensures all supplied parameters are treated as data, rather than potentially executable content. The use of stored procedures or prepared statements when interacting with databases will provide significant protection and ensure that the supplied input is treated as data. OWASP further recommends putting mechanisms in place to handle any possible errors, timeouts, or blockages that may occur during operation. All output, return codes, and error codes from the web application should be checked to ensure that the expected processing actually occurred. At a minimum, error handling should be implemented in a way that would allow the application to detect problems during the execution of the web application. The Advosys Consulting Group has another approach in dealing with this matter. They recommend logging suspicious errors. Certain types of error messages can indicate attempted attacks on a web application [5]. Without such error handling and reporting, injection flaw attacks can occur without ever having been detected.

An advisory, CERT® Advisory (CA-2000-02), has worked on “Malicious HTML (Hyper Text Markup Language) Tags Embedded in Client Web Requests” [6]. The advisory has devised a solution for Web Page Developers and Web Site Administrators for protecting their web sites. They have strongly advised developers to restrict variables used in the construction of web pages to those characters that are explicitly allowed and to check those variables during the generation of the output page. Also, web pages should explicitly set a character set to an appropriate value in all dynamically generated pages.

The OWASP Filters project supplies functions for sanitizing input information, but no actual filters, as the name of the project implies. This allows many interpretations as to how the filter will work and what the filter will be allowed to do.

There has been other research into eliminating injection flaws, many of which require some sort of outside filter to be used. The AMNESIA research focuses on an external program that will filter the code to create an indefinite automation and compare this to the actual results [10]. Another example that relies on a filter is the DOME research [11]. DOME uses a filter that looks for and marks the locations of system calls and then watches the result of the execution of the actual code. Moreover, WebSSari is another approach where a filter is used to both prepare and check the output [12]. These research concepts used some sort of filtering approach that is called outside of the execution of the code. We will be using similar filtering techniques, but they will be run as part of a normal routine when executing a web page command that the server will call.

We also specify the method of proficient usage of Filter Chain Operation in our methodology which increases the efficiency of the Filter Chains to a greater extent than using the approach traditionally.

3. INTERCEPTING FILTER APPROACH

This paper proposes a new approach for eliminating injection flaw exploitations, thus increasing the security and reliability of web applications. Our strategy, which relies on the use of automatic intercepting filters, will address the weaknesses of the current methods utilized for preventing injection flaws. We also provide the design for a detailed filter for preventing Cross-site scripting vulnerability.

As stated previously, our approach relies on intercepting filter components to eliminate injection flaw exploitations. A filter component dynamically intercepts HTTP (Hyper Text Transfer Protocol) Requests and HTTP (Hyper Text Transfer Protocol) Responses to transform, or use, the information contained in the HTTP (Hyper Text Transfer Protocol) Requests or HTTP Responses [7]. J2EE application servers, Java Servlet compliant containers, and ASP.NET application servers provide the built in filter functionality. The Java Servlet specification version 2.3 introduced the filter component type to Java. ASP.NET provides filter functionality through custom HTTP modules. An HTTP module is a class that implements the ASP.NET IHttpModule interface, and determines when a filter should be called [8]. Moreover, currently the **L**inux **A**pache **M**ySQL **P**erl/PHP (LAMP) stack does not provide filter functionality. However, it may be possible for the LAMP stack to emulate filter-based functionality using Apache based redirect rules.

The implementation of Java-based filters and .NET-based filters both follow the "Intercepting Filter" design pattern, which is a core pattern for both Java and .NET development. The Intercepting Filter pattern works in the following way. When the web server receives a resource request, such as a form submission, the web server's request handler passes the control to a FilterChain object. The FilterChain object maintains a list of all filters for the specified resource, and calls each filter in sequence. The FilterChain object can read the sequence of filters from a configuration file (the web.xml deployment descriptor file in Java-based web applications, and the web.config application configuration file in .NET-based applications) to achieve deployment-time composability. Each Filter object called by the FilterChain object has a chance to modify the incoming request. For example, Filter objects can modify the response URL (Uniform Re-

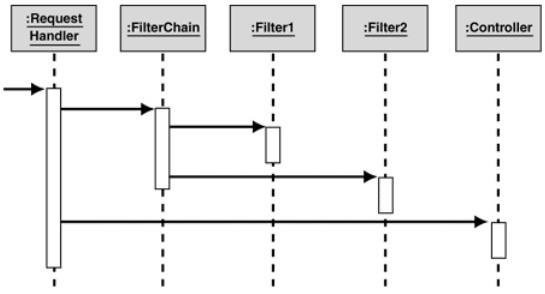

Fig. 1. Intercepting Filter Pattern

source Locator), or add header fields to be used by the web application. After all the Filters' objects in the FilterChain have been executed, the request handler passes the control to the controller, which then executes the web application functionality. Figure 1 illustrates the Intercepting Filter design pattern.

Our proposed strategy utilizes filter components in the "Intercepting Filter" design pattern to automatically sanitize incoming HTTP Requests to prevent injection flaws.

3.1 Filtering Method

Multiple filter components can be created to automatically sanitize each parameter in the incoming HTTP Requests. These filter components can then be chained together to provide complete coverage against injection flaws. Example 3.1 provides an outline for a Java SQL filter object, which can be used to eliminate the risk of SQL injection flaw exploitations.

The Java SQL Filter provided in Example 3.1 can be combined with other filter components, with each filter component designed to sanitize against a specific method of injection flaw ex-

```
package com.csc.filters;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SQLFilter implements Filter
{
   FilterConfig fc;

   public void doFilter(ServletRequest req, ServletResponse res
   FilterChain chain)
   throws IOException,ServletException
   {
      //get the HTTP Response
      HttpServletResponse response = (HttpServletResponse) res;
      //get the HTTP Request
      HttpServletRequest request = (HttpServletRequest) req;
      //iterate through the request parameters
      //and sanitize them

      for(Enumeration e = request.getParameterNames();
      e.hasMoreElements();)
      {
         //sanitation methods, not included in example
      }
      // pass the Request/Response on
      chain.doFilter(req, response);
   }

   public void init(FilterConfig  filterConfig)
   {
      this.fc = filterConfig;
   }

   public void destroy()
   {
      this.fc = null;
   }
}
```

Example 3.1 The object model

ploitation. The linkage of these filter components can be achieved through a FilterChain object, which would ultimately provide a wide coverage area of protection. For example, the Java SQL Filter in Example 3.1 can provide sanitation methods specific to SQL injection flaws, such as removing special characters or removing occurrences of "—" strings, which can be used to run arbitrary SQL code. Other filter objects can be created to sanitize HTTP Requests for other types of injection flaws, such as removing script language elements from input. Once these filters are created, they can be linked together by the FilterChain object,-thereby providing multiple layers of validation to any input before it is executed by the web application. One of the specific filter components used to combat the Cross-site scripting vulnerability is discussed below.

3.2 Cross-Site Scripting Vulnerability Filter

This filter component would be inside the above discussed filter object. As the supplied data reaches the server-side application components, input filtering can be done by removing some or all of the special characters from the data that is supplied by the user. Client side input filtering should never be relied on because the attacker can easily work around this filter. Therefore, the server side process will carry out the input filtering processes.

3.2.1 Filter Input Parameters for Special Characters

Input filtering can be achieved by positive filtering. In this method input filtering is implemented by selecting from the set of characters that are known to be safe instead of excluding the

```
int Filter_q(BYTE *ip, int len)
{
        static char vstring[] =

        "1234567890!@%-_=+:,./\

        abcdefghijklmnopqrstuvwxyz\

        ABCDEFGHIJKLMNOPQRSTUVWXYZ";

          BYTE *op = ip;

          int i;

          for(i=0; i < len; i++)
          {
                  if(strcmp(ip[i],vstring))
                          op++ = ip[i];
          }

          return strlen(op);
}

Given string:

Jo&*hn Ab$rah@a~m, ai?n^t wo#rk|ing..!

Filtered String:

John Abraham, ain't working..!
```

Example 3.2 Input Validation Code for numerical field

named special characters. Implementing positive filtering also helps in exploiting other unknown vulnerabilities.

For Example, let's suppose a form field expects a range of digits 0 through 9. This field cannot accept any letters or special characters. Consequently, when the following code is used for input validation, it will check all the fields as well as clearing out all other unwanted characters.

The code above works by comparing the user inputs with all the characters that the programmer has decided should be included in the filter. If the user input doesn't match with any of the pre-defined characters, then that character will not be outputted as a legal character.

3.2.2 Encoding / Decoding of Request/ Response

The positive filtering process discussed above has one disadvantage; it can lose data. If a client sends some confidential data in the request to the server, then after the filtering process, the data may change and become useless. Hence, it is essential to encode the user input in order to protect the malicious script from execution. This method ensures that the input stays unchanged while also disallowing the script to cause any damage. The following is the Encoding/decoding scheme.

Table 1. Encoding/decoding scheme

Input	***Code***
<	/lt
>	/mt
“	/dq
‘	/sq
%	/pc
;	/sc
{	/bp
}	/ep
&	/am
+	/ps
/	//

It also has to be noted that this table is not the only table that can be used as a reference for an Encoding/Decoding scenario. A different look for this kind of table can be found at cert.org [7] under its ISO 8859-1 (Latin 1) character set. The function encode_string is placed in the server when a request arrives. The filter encodes the input and passes the request to the controller. Then the function decode_string should be placed when the server sends the response back to the client. Or the code can also be placed before the user's browser gets the response to display.

One illustration is given in this example below

<SCRIPT> Hello </SCRIPT>

Encoding:

/ltSCRIPT/mt Hello /lt/SCRIPT/mt

```
int encode_string(BYTE *ip, int len)
{
	BYTE *op = ip;
	int i=0;
	for(i=0; i < len; i++)
	{
		switch (ip[i])
		{
			case '<': op++ = '/';
				op++ = 'l';
				op++ = 't';
				break;
			case '>': op++ = '/';
				op++ = 'm';
				op++ = 't';
				break;
			;
			;
			;

			default: op++ = ip[i];
		}
	}

	return strlen(op);
}

int decode_string(BYTE *ip, int len)
{
	BYTE *op = ip;
	int i=0;
	for(i=0; i < len; i++)
	{
		if(ip[i] == '/')	//if the char is '/' then check next two chars
		{
			i++;
			switch (ip[i])
			{
				case 'l':i++;	//if next char is 'l' check next char
					if(ip[i] == 't') //if its 't'
					op++ = '<';		// insert '<' in the output
				case 'm': i++;
					if(ip[i] == 't')
					op++ = '>';
				;
				;
				;
				case '/': op++ = '/';	//if next char is '/' again insert only one '/'
				default: op++ = ip[i];
			}
		}
		else
		{
			op++ = ip[i];
		}
	}
	return strlen(op);
    }
```

Decoding:

<SCRIPT> Hello </SCRIPT>

Once again, the advantage of an encoding/decoding scheme over positive filtering is that it won't lose any data. One similarity is that the filter in the encoding/decoding method takes the character that the user inputted and manipulates that into the code, based on table 1. However, the difference is that this filter also allows the user to input any characters, which are not on the table, and manipulate them.

3.3 FilterChain Operation

The FilterChain object maintains the list of Filter objects specific to the resource type. Furthermore, each Filter object in the list can be provided classification to reduce the overhead of executing all the filters for each request. Functionality can be added to the FilterChain object to identify a type of content and select the filters accordingly.

A FilterChain object can select the filters for HTTP request using the following methodology:

1. If the content is static then no filter is selected.
2. If more than one filter type is needed for a particular content, then all the involved types of filters are selected.

There may be an enormous number of filters added to the FilterChain object pertaining to the many common security holes on the internet. Going through each filter before passing the information to a web application may require great amounts of time which becomes unacceptable in terms of performance. For this reason, the following additional criteria should be added to select the limited number of filters from the available list.

1. A timestamp can be added to the filter header which will allow the newest filters to be selected first. This is because; an old vulnerability may have already been fixed by a web application with the filter for which having yet to be removed from the chain.
2. A priority field can be added to the filter header which would allow a low priority (or recoverable vulnerability) filter to be dropped in case of a huge number of filters.
3. The number of filters to be included in a filter chain may also depend upon the amount of load in an available system. For example, a busy system may select only the first five filters after applying the criteria while a fairly vacant system may select ten filters for its chain.

4. ANALYSIS AND LIMITATIONS

The Intercepting Filter pattern approach to prevent injection flaw exploitations has a number of advantages over the existing methods discussed in section 2. For example, a filter component can be used to automatically sanitize all data in an incoming HTTP Request before it is passed to the web application. Such a filter would reduce the size and complexity of the web application's source code. Automatic filtering would eliminate the need for numerous individual functions to call upon sanitizing methods for every parameter being used by the function, thus reducing the amount of code needed for each function. Additionally, automatically filtered web applications would also be easier to maintain, since the entire sanitizing process is automatically done

at a centralized location, rather than relying on decentralized objects calling upon multiple sanitizing functions. Automatically filtered web applications would also be more secure than non-filtered web applications. If a web application is automatically filtered, then there is no need for programmers to have to remember to call upon sanitizing functions for each parameter being used by a function. Eliminating this source of potential human error would eliminate the risk of un-sanitized parameters being used during web application processing. In addition to using the filters that are already provided by the languages used, the maintainers will be working with known filtering methods instead of the almost black box type of sanitizer provided by a 3rd party.

In addition to these benefits, the implementations of the Intercepting Filter pattern are extremely flexible. For example, a portion of a web application that only serves static content probably does not need to be filtered. Configuration files can be deployed with the web application specifying that the static portions are unfiltered, which would conserve server processing power. Additionally, if a specific portion of a web application needed additional filtering, the web application could be configured so that specific portions receive additional filter components. For example, if only a portion of a web application is inserting records into a back-end database, this portion of the web application can be configured with an additional SQL sanitation filter.

5. Conclusion and Future Work

With the growing omnipresence of web applications in the global economy, it has become critically important to develop secure and reliable software to support the economy's growing dependence on web-based systems. Due to the "highly available" nature of web applications, they provide a large surface area for potential vulnerabilities. These vulnerabilities are often exploited and used to attack web applications. Perhaps the most critical of these web application vulnerabilities are injection flaw exploitations.

The proposed Intercepting Filter Approach can greatly mitigate the risk of injection flaw exploitations. The use of filter components, in conjunction with the Intercepting Filter design pattern, can be used to sanitize HTTP Request information before it is ever processed by the web application. The implementation of the proposed approach can be carried out on Java and .NET based platforms. It is also possible to emulate a filter on the LAMP stack.

There are a number of possible avenues of future work available for this paper. The specific filters needed to provide protective coverage against known forms of injection flaw exploitations were not discussed. Another issue not addressed by this paper is that only one actual sanitation method was discussed, but there may be as many as fifteen different individual filters needed to protect against each specific type of injection flaw. When adding multiple filters you may acquire the need of some form of Artificial Intelligence system to handle which filters are used and for what case or rule based on the input fed to the filtering sub-system. For example, an SQL filter may need over fifteen sanitation methods to ensure that the input is safe for execution. In addition, error logging methods, which were not implemented in this study and are one of the important measures considering the security of websites and servers, could be an extension of this paper. Another possible area of future work could include a project to implement reusable filter components in Java or .NET environments. These filter components could be checked

against security analysis software to confirm that the Intercepting Filter Approach eliminates injection flaw exploitations.

REFERENCES

[1] OWASP.org, the OWASP Top Ten is a list of vulnerabilities that require immediate remediation, http://www.owasp.org/documentation/topten/introduction.html

[2] SPI Dynamics Inc, SQL Injection White Paper, SPI Dynamics Inc., 2002.

[3] Advisees Consulting Group, Writing Secure Web Applications, Advisees Consulting Group, 2004.

[4] CERT® Coordination Center, CERT® Advisory CA-2000-02, Malicious HTML Tags Embedded in Client Web Requests, CERT Coordination Center, Carnegie Mellon University, Pittsburgh PA 15213 -3890, USA, 2000.

[5] Duffy, Kevin, et al., Professional JSP Site Design, Wrox Press, 2001.

[6] Anderson, Richard, et al., Professional ASP.NET 1.0, Wrox Press, 2002.

[7] CERT® Coordination Center, Understanding Malicious Content Mitigation for Web Developers, CERT Coordination Center, Carnegie Mellon University, Pittsburgh PA 15213-3890, USA, 2000.

[8] Ollmann, Gunter, Understanding the cause and effect of CSS (XSS) Vulnerabilities, http://www.technicalinfo.net/papers/CSS.html

[9] W. Halfond and A. Orso, Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks, *Proceedings of the Third International ICSE Workshop on Dynamic Analysis, WODA 2005*.

[10] W. Halfond and A. Orso, AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks, *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering, ASE 2005*.

[11] Rabek, Jesse C., et al, Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code, Defense Advanced Project Agency (DARPA), Copyright Association for Computing Machinery, ACM, 2003.

[12] Huang, Yao-Wen, et al, Securing Web Application Code by Static Analysis and Runtime Protection, New York, New York, USA, 2004.

[13] Jerry Lee Ford, Jr and William R. Stanek, Increase Your Web Traffic, fourth edition, Thomson Course Technology, 2006.

[14] Joel Scramby, Mike Shema and Caled Sima, Hacking Web Applications Exposed, second edition, The McGraw-Hill Companies, 2006, pp.238.

[15] Stuart McClure, Joel Scramby and George Kurtz, Hacking Exposed, Network Security Secrets & Solutions, fifth edition, The McGraw-Hill Companies, 2005, pp.581-582.

[16] Hackademix website-http://hackademix.net/2007/08/12/united-nations-vs-sql-injections/

Ahmed Salem

Dr. Ahmed Salem is an associate professor of computer science at California State University, Sacramento (CSUS). Dr. Salem also is the director of the Software Engineering Research Center (SERC) at California State University, Sacramento. In this capacity he conducts research and training in the area of software verification/validation, software development, requirements modeling and design, and software quality assurance. Dr. Salem also worked as a research engineer at the University of Washington, Seattle WA. Dr. Salem serves as a consultant for various state agencies as well as private industry. Some of which include, Texas Instruments (TI), Department of Fish and Game, California Department of Forestry and Fire Protection, California Department of Transportation (Caltrans), California Department of Corrections (CDCR), HP, and Microsoft, Dr. Salem has been on program/organizational com-

mittees of several software engineering and computer science international conferences. He also served as a reviewer for a number of software engineering books. Dr. Salem is a member of the Software Quality Association, the Technical Council on Software Engineering – IEEE Computer Society, the Shared Software Infrastructure Program (SSIP), and the Software Process Improvement Network (SPIN) Organization. His research interests includes: software engineering, software testing and quality assurance, software verification/validation, software requirement and design modeling, software reliability, software process improvements, and software engineering education.