

A Short Guide to Input Validation

secologic Whitepaper

25.04.2007 Version 1.0

Table of Content

1 Introduction..... 3

2 Basic validation approaches 4

 2.1 No input validation 4

 2.2 Blacklisting..... 5

 2.3 Whitelisting 6

 2.4 Sanitizing..... 7

 2.5 Mixed validation strategies..... 7

3 Current architectures for input validation 8

4 Guidelines for input validation..... 10

 4.1 Validate all input 10

 4.2 Use your framework or IDE for the monkey code..... 11

 4.2.1 STRUTS input validation..... 11

 4.2.2 ASP.NET Validator Controls 12

 4.3 Test the input validation 13

 4.4 Last advice: ask a security expert to secure high security applications 13

5 References..... 14

1 Introduction

The objective of each business application is data processing. Some input is processed by some rules given by the program and this generates some output. If a hacker sends input to the application which is not as well formed as expected than the input can be processed incorrectly and reveal protected resources. A classical firewall can defend your application from attacks performed at lower OSI layers but it can not protect against attacks by the use of malicious input. Thus there is a need to protect your application from invalid input.

Input validation is effective because this countermeasure covers several of the most common web application vulnerabilities. According to the OWASP list of the *Top Ten Most Critical Web Application Security Vulnerabilities*¹ the items “A1 - Invalidated Input”, “A5 - Buffer Overflows”, and “A6 - Injection Flaws” can be mitigated by input validation.

Input validation is no new invention of security experts. It has been a best practice from the beginnings of programming to verify all data syntactically. Unfortunately in the times of “rapid prototyping” and “time to market” pressure this guideline has been lost widely. Even some modern development environments does not support input validation in a convincing way. Thus programmers have to find their own way of input validation.

From a scientific point of view, input validation is a semantic problem. Input will be accepted and processed by a program. The input is an element of an object language. The program itself uses a meta language (script code, assembler code, etc) to control the processing. The objective of an attacker is to deliver object language which partly will be interpreted as meta language.

We discuss several input validation solutions and present best practices and basic examples in Microsoft's ASP.NET and in Java's struts framework.

¹ <http://www.owasp.org/documentation/topten.html>

2 Basic validation approaches

In this section we describe the basic approaches how to process incoming input.

2.1 No input validation

In general it is not recommended to skip validation. There is an exception of this rule. If the design ensures that the input has been successfully validated before then it is acceptable to skip the input validation. However as a basic principle each component should be able to defend itself. A component may skip input validation only if it has a trust in the correctness of the input validation of the delivering component. This is a classical example of the *design by contract methodology*. Each component has preconditions which must be satisfied by the caller. In this case the precondition would be “all input delivered to the called function has been successfully validated by the calling function”. This assumption must hold during over the complete software (component) lifecycle. In particular reuse of components is susceptible for the violation of mutual contracts.²

1. The change management in the software (component) lifecycle must guarantee that all changes of the contract must be inherited to all contracts with dependable modules.
2. The validating function must “know” what malicious input is for all subsequent callable functions.
3. New callable functions in the backend can require changes in the input validation of the calling functions.

These are three strong requirements which are difficult to assure over long times. Furthermore such strong requirements between functional unrelated modules are undesirably because they make difficult the decoupling of components.

Best practice: It is not recommended to skip input validation at interfaces to untrustworthy sources. No input validation is acceptable for internal function and methods which can trust their callers. In particular private methods can expect trustworthy input which may not be validated again.

² One of the most famous examples is the crash of the first Ariane 5 rocket in 1996 . A control system which was designed for Ariane 4 expected only a limited range of input data. Ariane 5 is much taller and heavier and the input data infringed the former “reasonable” assumptions. See **Computer-Related Risks**, Addison-Wesley/ACM Press, ISBN 0-201-55805-X, 1995, for details.

2.2 Blacklisting

The term “blacklisting” means to filter evil input only. Unfortunately “evil input” can not be exactly defined. Usually blacklisting is based on the detection of possibly or probably or certain malicious input. The most common example is filtering of the JavaScript tag `<script>` to avoid Cross Site Scripting. This inhibits code injections like

```
<script>alert("You have been hacked!")</script>.
```

Blacklisting does not enjoy a good reputation. There are some inevitable problems which have no satisfying solution.

1. One can not define all evil input. Only known attacks can be described in patterns. New attacks must be detected, analyzed, described in corresponding patterns, and afterwards implemented. This is the same awkward position of the antivirus industry: “You can only defeat what you know.”
2. A famous motto of Perl programmers is “There’s more than one way to do it.”³ This holds for the bad guys too. They are very clever to develop circumventions of blacklists. For example an ASP.NET page should throw an exception if an input form field is filled with “`<script>`”. In ASP.NET 1.1 this doesn’t hold for “`<%00script>`” but browser render both strings in the same matter such that an attacker can use `<%00script>` to perform a XSS attack.

Nevertheless, blacklisting is very popular. It can partly defend against specific known threats. Sometimes it is the only possibility. If your application server has a well known but not yet patched vulnerability then blacklisting the attack at your bastion host can be an appropriate countermeasure.

Some vendors have implemented default blacklists. For example in ASP.NET all requests are checked with a hard coded list of malicious strings.⁴ Obviously this concept is deficient by design but it could be the starting point of a “poor man’s blacklist”.

Sometimes Blacklisting is an appropriate approach. It can be used to fill the gap between the release of a so called “Zero Day Exploit”⁵ and the release of a well tested patch. Often such an exploit is identified by a characteristic attack pattern which can filter. This is not a solution for ages because later exploits can and will vary. However blacklisting can be a countermeasure to gain time for your devel-

³ Perl Docs. tbd

⁴ As long as the ValidateRequest attribute is active.

⁵ “Zero Day Exploits” are vulnerabilities which can be published simultaneously with an attack routine. In general an appropriate vendor patch or workaround is not available immediately.

opers fixing the real problem. A similar motivation for blacklisting is defer patching to a regular maintenance slot.

Sometimes an implemented blacklisting will be used as a replacement for a whitelisting. Only for the sake of completeness we remark that you should consider blacklisting only as a workaround of inferior quality.

Best practice: Avoid blacklisting if you can do better. Whitelisting is preferable from the viewpoint of security. However blacklisting is much better than no input validation.

2.3 Whitelisting

Whitelisting is the recommended best practice for input validation. All input may only pass if it is validated as known good input. This is also called positive input validation in contrary to the negative validation of the blacklisting. This is strong requirement and in real world application not easy to implemented. The simplistic standard example is the positive validation of a ZIP with a plain regular expression. Example: the regular expression

`^\d{5}$`

can be used to validate a German zip code consisting of five digits.

The basic idea of whitelisting is that validated input is classified as non-malicious.⁶ There is a consensus in the community to consider it as the most secure approach. On the other hand the experiences from security audits show that sufficient implementations are not very common. There are some reasons for that.

1. A successful whitelisting requires a very exact knowledge and rigid formal definition of the business data. In the best case there is a formal specification of the expected input. Otherwise they must be provided in the business requirements of the application. Such detailed documents can be an expensive and require time consuming discussions. Changes in the requirements cause usually changes in the input validation and test cases.
2. The description patterns for whitelist entries are usually regular expression. It is not efficient to code regular expression one-to-one to input fields. Thus there is a need for an input validation design pattern or framework.
3. There is no classical design pattern for input validation. The design of a strict and comprehensive whitelisting is difficult. For example there are software components which are only used to transmit data. They can not decide for themselves what is valid or not with respect to following components. A validation in front of these components mirrors some parts of the business logic in the validation logic. This is a great effort, difficult to maintain and collides pos-

⁶ You are in trouble if syntactically validated input can be malicious to your system. In that case you have been run into an design flaw which can not be mitigated by simple input validation. Maybe sanitizing could be a workaround.

sibly with the separation of tiers in multi tier architectures. The responsibility for input validation shifts from business logic to gateway components. Errors and false positives can be hard to track in complex software projects.

4. It is easy to test blacklisting with known attacks. It is more difficult to test whitelisting against false rejections and against false acceptance in a rigid way.

Best practices:

1. **Use whitelisting whenever possible.**
2. **Check the input length every time and restrict it to the shortest possible value.**

2.4 Sanitizing

Sanitizing is the approach to defuse a possibly malicious input and to replace it by non malicious input. Example: you have a web application which is vulnerable to SQL code injection but there is some reason that you can not fix the problem at the data access layer. A simple and strict whitelisting excludes all SQL meta characters is not favored because (for example) the name "O'Malley" would be rejected. Obviously "O'Malley" is a legitimate and non malicious input. A sanitize filter could omit or replace the apostrophe with a description. Example

"O'Malley" will be transformed to "OMalley" or

"O'Malley" will be transformed to "O[apostroph]Malley".

Both possibilities damage the integrity of the input but that is the purpose of sanitizing. Both possibilities are human readable thus they could be completely sufficient for a customer care agent for example. The second one is reversible which can be helpful for some purposes.

Best practices:

1. **Avoid sanitizing if possible.**
2. **If you can not avoid sanitizing do it first and whitelist it second.**

2.5 Mixed validation strategies

In real life the situation is often more complex. Usually a real life application comprises several components, some of them are self developed products, and other components are black box components by miscellaneous vendors. In such cases one can use a mix of the approaches above. Here we recommend to ask an experienced web application security consultant.

3 Current architectures for input validation

Input validation can be done with many different approaches. The most basic way to validate input is to call explicitly a regular expression matching routine. This simple implementation can be sufficient in small projects but it lacks of checking for completeness and maintainability.

A better solution is centralized validation function or module. But both approaches require the possession of the sources.

Another method is the filtering of the data stream before the reaches the application. This can be done inside the hosting web server or by dedicated reverse proxy web servers or appliances. The major advantage of these external validations is the “black box” capability. It can be implemented afterwards and for closed source software. However, the training of such an adaptive filter is challenging with respect to the false positive and false rejection rates.

In the following table we summarize the most important characteristics.

Validation method or component / Evaluation criteria	Validation in front of the application and before the input data will arrive the web server	Inside the web server but before the "coded" part of the application	Inside the application with central modules	Within the code "form by form"
Examples	Web application firewalls (WAF)	<ul style="list-style-type: none"> WAF web-server plugins Apache's mod_security 	Using the validation routines in Struts	Using validation controls in ASP.NET
Typical purpose	Protection of Common of the Shelf (COTS) web applications	Protection of COTS web application but one has control about the webserver	Self developed applications based on a framework or with validation services	Selfdeveloped applications based but not based on a framework
Individual advantages of such a solution	A web application firewall (WAF) can implemented independently of the underlying business application. There are no (or only few) changed necessary.	Similar to the WAF scenario but one can spare an additional proxy. This improves performance, maintaining and costs.	Central validation control point. It is easy to check that all input will be validated. Changes in the whitelist can implemented at a single point.	Easy to implement. Sufficient and rigid approach for small and middle size applications.
Individual disadvantages of such a solution	<ul style="list-style-type: none"> Requires additional hardware and software Reduces performance and stability Needs extensive training of the WAF No programmatic interaction with the application. 	<ul style="list-style-type: none"> Increases slightly the complexity of the workflow Reduces slightly performance and stability Needs extensive training of the filter. No programmatic interaction with the application. 	<ul style="list-style-type: none"> Only applicable when such frameworks are used. There is no well accepted design pattern for input validation. 	<ul style="list-style-type: none"> It is difficult to verify the completeness and rigidity of the validation. Changes of business data specification can result in complex refactoring.
Occurs for the first time in which phase of software lifecycle	<ul style="list-style-type: none"> Testing or Production 	<ul style="list-style-type: none"> Testing or Production 	<ul style="list-style-type: none"> Design 	<ul style="list-style-type: none"> Implementation
Performance issues	Moderate: "yet another proxy"	Good	Good	Good
False positive rate	Annoying	Annoying	Low	Low
Request specific error handling	None	None	Possible	Possible
Protected components	<ul style="list-style-type: none"> Application Web server 	Application	Application	Application

Table 1 Characteristics of popular input validation approaches

4 Guidelines for input validation

4.1 Validate all input

This is the most important rule. There are many input types which can be used to attack an application:

- GET parameters
- Forms fields (hidden fields)
- Selection lists and drop down lists
- File Upload (file name and file content)
- Cookies
- HTTP-headers
- Java applet communication
- Flat file import
- ...

GET parameters and form field are the most obvious inputs for web applications. Also back posts for selections lists and drop down lists must be validated to avoid SQL injections for example.

It is not obvious how to validate file uploads in a perfect manner. At least the pathname must be handled with special care. The file extension should be restricted to a reasonable and short list of acceptable file types. The validation of the path component of the file name must prevent arbitrary access to the file system (example “c:\windows\system32\trojan_horse.exe”) and path climbing (example “..\..\..\system32\evil_driver.dll”). Usually the correctness of the content of a file is very difficult to verify because the specification of file formats are complex or even unknown (for example .pdf or .ppt).

In general an application should avoid writing and reading of own specified cookies. The use of cookies for session management is sound but this is a task for the application server framework. Usually these frameworks support the saving of session parameters as key value pairs in a session object in an easily manner. This object is stored internally at the server and not at the client. Thus there is no need to validate keys and values at every request. In contrast a validation has to be performed if session parameters are stored in cookies or hidden field. Transmitting session data to the client increases the threat of data spoofing. You can find more information about best practices in session management in a white paper by Secologic⁷ or in the OWASP Guide⁸.

⁷ Web Application Session Management, secologic Whitepaper, 2007
<http://www.secologic.de/>

⁸ The OWASP Guide to Building Secure Web Applications,
http://www.owasp.org/index.php/OWASP_Guide_Project

There are two important exceptions for the strict rule of input validation. If a module hands over input one to one to another module then it can skip input validation. However the module must protect itself.

The second exception is input from trusted sources. The notion of a “trusted source” is a controversial issue. It requires extensive control about the source. The trust in a source can possibly change in the life cycle of the source or by re-use of the component. In general we do not recommend this concept. In any case a trusted source requires a diligent threat modeling⁹.

4.2 Use your framework or IDE for the monkey code

A developer should not spend too much time with input validation coding issues. Frameworks like STRUTS or ASP.NET support developer friendly input validation. We present two examples here:

4.2.1 STRUTS input validation

In the following example (adaptation of the original documentation¹⁰) you can see how input validation in STRUTS works. The code is nearly self explanatory.

```
<field property="lastName" depends="required,mask">
  <msg
    name="mask"
    key="registrationForm.lastname.maskmsg"/>
  <arg position="0" key="registrationForm.lastname.displayName"/>
  <var>
    <var-name>mask</var-name>
    <var-value>^[a-zA-Z]*$</var-value>
  </var>
  <var>
    <var-name>maxlength</var-name>
    <var-value>32</var-value>
  </var>
</field>
```

Example 1: Input Validation in STRUTS. “STRUTS Validator Guide: <http://struts.apache.org/struts-action/faqs/validator.html>”

The dependency “mask” introduces the input validation with a regular expression. The crucial tag is “<var-value>” which contains the regular expression “^[a-zA-Z]*\$” describing the syntax of positive acceptable input.¹¹ In particular the size of the input should be restricted to a reasonable small value, say 32 in this example.

⁹ See „Threat Modeling” by Frank Swiderski and Window Snyder, Microsoft Press, 2004 or <http://msdn2.microsoft.com/en-us/security/aa570411.aspx>.

¹⁰ Struts validator Guide, http://struts.apache.org/1.2.4/userGuide/dev_validator.html

¹¹ Remark: this example has been taken from the original documentation of STRUTS for demonstration only. Obviously the regular expression “^[a-zA-Z]*\$” is not very useful because it lacks localization.

The concept of input validation in STRUTS is very convincing. All validations can be configured in a central module or file. This is easy to review for rigidity and completeness, easy to test and easy to maintain.

An improvement in STRUTS is the introduction of variables for regular expression in the validation module. We cite an example taken from the struts documentation.

```
<global>
  <constant>
    <constant-name>zip</constant-name>
    <constant-value>^\d{5}(-\d{4})?$</constant-value>
  </constant>
</global>

<field
  property="zip"
  depends="required,mask">
<arg0 key="registrationForm.zippostal.displayname"/>
<var>
  <var-name>mask</var-name>
  <var-value>${zip}</var-value>
</var>
</field>
```

Example 2: Using a variable for an regular expression

The usage of variables for regular expressions enables very efficient validation solutions. For example a business data dictionary could contain all business data types with their regular expression and the name of their variable. A developer can reference to these variables names for validation purposes.

4.2.2 ASP.NET Validator Controls

Microsofts ASP.NET contains very powerful validators for input validation which can be easily integrated in their development environment VisualStudio 2005. In particular, the RegularExpressionValidator is a useful tool from the viewpoint of security.

```
<asp:RegularExpressionValidator
  ID="RegularExpressionValidator1"
  runat="server"
  ControlToValidate="txtUserName"
  ErrorMessage="Username is not valid."
  ValidationExpression="\w{1,10}">
</asp:RegularExpressionValidator>
```

Example 3: Input validation in ASP.NET using the RegularExpressionValidator control

The validation expression in this example allows all usernames with one up to ten word characters (“\w”).

There is a pitfall while using ASP.NET validators. The binding of a validator control to a form control does not change the execution flow of the page at the server side. The only effect is the setting of the “IsValid” property of the respective validation control. The developer must query this property to control the execution of the page. Additionally the validation of the client side JavaScript code can pretend a secure validation while the server side validation is still missing.

ASP.NET validation controls validate input mandatory at the server side but they can generate additional client side JavaScript validation code. Unfortunately, the interpretation of regular expressions in JavaScript is slightly different from the interpretation in more powerful regular expression server machines. In particular, internationalization and Unicode causes serious troubles. We recommend to tailor the validation by client side JavaScript manually in complex situations.

4.3 Test the input validation

It is very important to test your input validation. We give here only short list of different aspects in validation testing:

- Test for false positives: Your business must not generate validation errors.
- Test for false negatives: Test with invalid data to check the correctness of the validation routines.
- Test with and without JavaScript: As described above JavaScript interprets regular expression slightly different compared with server engines. These differences can generate false positives at the client side.
- Perform a test **without** any validation at the client and the server. This test can expose insecurities in other tiers of the application. Some developers rely on the correct validation of the application frontend which is not sufficient.

4.4 Last advice: ask a security expert to secure high security applications

Best practice papers are useful as introduction and general guidelines. Nevertheless, they cannot meet all requirements of complex and challenging projects. In particular we recommend ask an experienced web application security expert to develop high security applications.

5 References

- [1] The Ten Most Critical Web Application Security Vulnerabilities, The Open Web Application Security Project, 2004, http://www.owasp.org/index.php/OWASP_Top_Ten_Project
- [2] Session Management, secologic Whitepaper, 2007, <http://www.secologic.de>
- [3] A Guide to Building Secure Web Applications and Web Services, The Open Web Application Security Project, 2006, http://www.owasp.org/index.php/OWASP_Guide_Project
- [4] Sicherheit von Webanwendungen, Maßnahmen und Best Practices, Bundesamt für Sicherheit in der Informationstechnik, 2006, <http://www.bsi.de>
- [5] STRUTS Validator Guide, 2000-2004, The Apache Software Foundation, http://struts.apache.org/1.2.4/userGuide/dev_validator.html
- [6] Threat Modelling, Frank Swiderski and Window Snyder, 2004, Microsoft Press