

Security Pattern for Input Validation

Lars-Helge Netland, Yngve Espelid, Khalid Azim Mughal
Department of Informatics, University of Bergen
{larshn, yngvee, khalid}@ii.uib.no

Abstract

This paper discusses a security pattern for input validation in web applications. A template description facilitates understanding of the important concepts, and allows developers with a security background to quickly adapt the pattern in their own context.

1 Introduction

E-commerce has become a major factor in the marketplace. In Norway, the percentage of total turnover from E-commerce grew from 2.2% in 2002 to 3.9% in 2005 [1]. This rapid growth has fueled a need for new software for selling goods and services over the Internet. In creating such systems, security aspects have been set aside in favor of time-to-market concerns and new application functionality. As a consequence, massive quantities of sensitive information lie inadequately protected on the Web.

Enter the villain. The Internet is a candy store for computer literate criminals. Snacks are readily available in form of credit card numbers, classified corporate data, and sensitive medical information. A long-time favorite target among hackers is the web-based shopping cart. Many developers use hidden fields to store and camouflage data from users, and fail to realize that the information can be easily manipulated. Through such tampering attacks, attackers can dictate their own prices in the web store, and hence purchase goods at extreme discounts. This vulnerability was reported in public no later than February 2000 [2], but software developers continue to release applications that are wide open to such attacks [3].

Software security is not a new research area. Saltzer and Schroeder's pioneering work "The protection of information in computer systems" [4], describes basic principles that remain true even today. More recently, much effort has gone into describing how attackers break software. In terms of building secure software, best practice guidelines have dominated the security literature. Patterns have become a popular way of sharing structured knowledge and experience from successful projects in traditional software engineering. A similar approach looks promising for distributing software security knowledge. The first paper in this direction was written by Yoder and Barcalow [5] in 1997.

Input handling is the process of checking data supplied by others against a set of predefined rules. The Open Web Application Security Project (OWASP) [6] has defined 3 categories of input handling:

Integrity checks, ensure that data has not been tampered with.

Validation, a set of rules that make sure that data is of a certain type, has correct syntax, is within specified length boundaries, or contains only permitted characters.

Business rules, checks that enforce back-end business logic, such as disallowing due dates already passed in an online banking application when paying bills.

The Input Validator pattern described in this paper addresses *validation* and *business rules*. It should be noted that specialized attacks such as SQL injection and cross-site scripting can be dealt with more effectively through prepared statements and HTML encoding, respectively [7]. It is possible to design a filter to look for metacharacter attacks, but identifying valid input (positive validation) is considered better than looking for data that can do you harm. Choosing the latter strategy is more costly in terms of maintenance, where you have to be constantly up-to-date with new threats, while the former option can provide protection against some of the attacks to come.

Unvalidated input is number one on OWASP's top ten list over web application vulnerabilities [8]. The rest of the paper presents a security pattern for input validation. We follow the template used in [9, p. 9], which is the de facto standard for writing security patterns.

2 Input Validator

A number of attacks on web applications target the application protocol through which vendors conduct business. Firewall technology and a PROTECTION REVERSE PROXY [9, p. 457] can enforce access control rules, but usually do not stop exploits embedded in message content. The back-end server(s) needs an INPUT VALIDATOR to validate data sent from the client.

Example

An Internet banking provider has recently been plagued by a series of hacker attacks. It started a few weeks back when about a dozen of the bank's clients filed reports demanding to know why their accounts had been cleared out. In the investigation that followed another problem was discovered: a fair quantity of bills paid by customers contained negative amounts. Fortunately, the bank invested heavily in a highly praised logging system a few months back, so sorting out the details should not be a problem. Section 2 describes how the bank improved their system's security.

Context

The input validator pattern applies to services offered through a client-server model communicating over HTTP. The goal is to validate input from the client. Input validation does not make network security solutions obsolete. It is a supplementary defense mechanism that helps to secure the environment in which the system runs. Any so-called client-side validation checks are worthless from a security standpoint. They serve the following three purposes: a) Enhance user experience through instant feedback on non-conforming input, b) conserve network bandwidth by reducing requests to the server, and c) save processing resources on the server. The term client-side validation is misleading, as none of the benefits mentioned above are related to security.

Problem

Many software systems fail to inspect and filter untrustworthy input. Through cleverly crafted input hidden in message content, attackers can alter the program logic and possibly exploit implementation weaknesses for economic gain. The pertinent question to ask is “How can one defend client-server applications against attacks hidden in the message content?”

Forces

The input validation solution must resolve the following forces:

- Extensive examination of input means more computational overhead. Developers should strive to make input validation as non-intrusive as possible. This is especially so in data-intensive systems, where delays that are acceptable in smaller applications, get compounded and render the system useless.
- The constant tug of war between software developers and malicious hackers has repeatedly shown how a dedicated attacker community discovers new ways of exploiting software. Systems should be designed with this ongoing battle in mind and incorporate flexible solutions that can be easily extended, if necessary, to counter future threats.
- Usability is a key factor for security constructs to prevail. Systems that are complex and difficult to operate cause maintainers and users to do mistakes that jeopardize security.

Solution

Perform syntactical and semantical validation of all input that is to be processed on the server. These 2 categories of validation are termed “validation” and “business rules” by OWASP. This process entails constructing appropriate validation rules for each source of input.

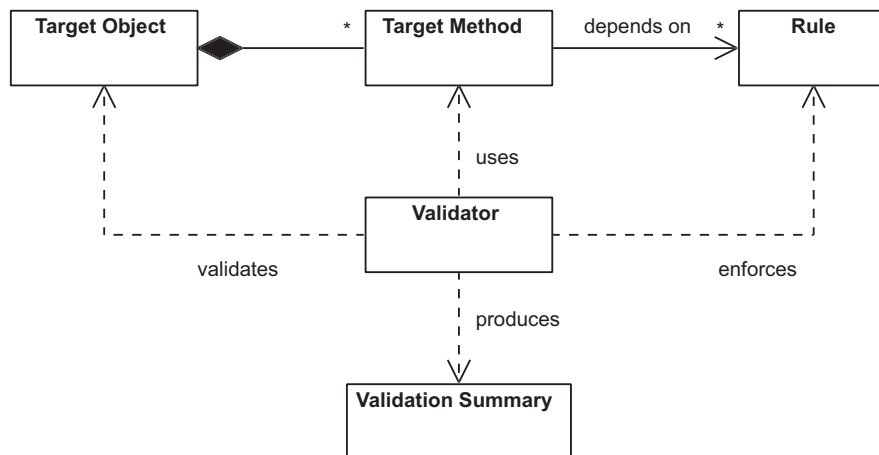


Figure 1: Content validation entities

Structure

Fig. 1 shows the class diagram for this pattern. The primary entities are

Target Object, which is the target for the validation process, comprising properties subject to inspection.

Target Method, which exposes an object property.

Rule, which defines constraints for the exposed object properties.

Validator, which controls the validation process, validating the exposed properties of an object against sets of rules and reporting any rule violations.

Validation Summary, which provides a summary of all rule violations.

Dynamics

The workings of an INPUT VALIDATOR can be explained in terms of an airport security checkpoint. Fig. 2 shows a typical setup, where travelers must pass a metal detector frame, and carry-on items are inspected in an X-ray machine. In this context, the primary entities can be interpreted as follows: The **Validator** is the airport security personnel and equipment; **Target Objects** are represented by airline passengers; **Target Methods** correspond to individual inspection targets, such as briefcases, cameras, or laptops; **Rules** express which items should be disallowed, typical examples include knives, guns, and can openers; **Validation Summaries** would be incident reports produced by the airport security personnel in the event of rule violations.

Consider the sequence of steps executed when a passenger shows up at the security checkpoint. First, the commuter must show a passport and valid travel documents to prove that he or her has purchased a service from one of the airlines

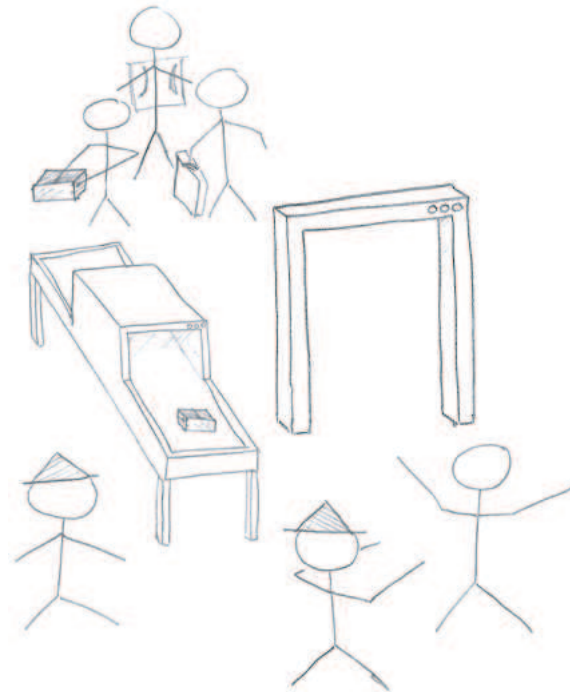


Figure 2: Passenger inspection at an airport security checkpoint

inside the security boundary. This is a precondition for input validation, and can be thought of as a firewall mechanism. Next, the security officials examine the traveler for illegal items, in accordance with a predefined list of disallowed objects. The passenger walks through a metal detector, while clothes, luggage, and other accessories are inspected with X-rays. If the airport security personnel finds it necessary, additional measures such as bomb-sniffing dogs and full body searches may be used. Travelers that do not carry prohibited items can continue their journey, while perpetrators are guided away from the security checkpoint and face further inquiries. Personnel at the checkpoint report offenses to other institutions, such as the police. Likewise, the INPUT VALIDATOR generates a `Validation Summary`, which in turn is handled by the application.

The process of explicitly specifying illegal items is more commonly referred to as *blacklisting*. This is a useful technique when you have a complete overview of the potential enemies of your system.

Implementation

Fig. 3 shows an example implementation of the INPUT VALIDATOR using firewalls [9, p. 403] and an INTEGRATION REVERSE PROXY [9, p. 465]. To implement this pattern, the following tasks should be performed

1. Identify all potentially vulnerable subsystems. Developers must single out components where message content will be processed. In Fig. 3, the stock application and the account manager can be exploited.

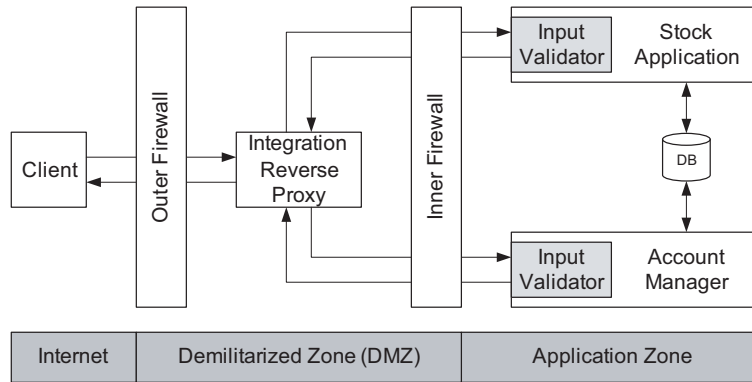


Figure 3: Example content validator architecture

2. Determine the entire set of client input sources and define what is to be considered valid input. We recommend using a whitelist approach [7, p. 71] to specify the format of valid input, also known as *positive validation*. All data not conforming to the format should be discarded.
3. Configure target objects and methods for the validators using the set of client input sources. Construct rules from the valid input definitions from the previous step, and map these rules to the corresponding target methods.
4. Deploy the validators. As shown in Fig. 3, validators should be integrated into every potentially vulnerable web application.

The input validator pattern should be combined with an accounting mechanism which can capture and store information about rule violations, and enable a reviewer to identify the involved participants and reconstruct the events that led to the violations. SECURITY ACCOUNTING REQUIREMENTS [9, p. 360] can help in selecting an appropriate accounting mechanism.

With a focus on functionality and user experience, many web application developers try to scrub content on behalf of the client. If violations are reported during validation of a newly received request, programming logic tries to alter the input with the goal of making it valid. Often, this scrubbing logic is complex and may incur significant processing overhead. A better alternative is to use the logs to evaluate the validation rules, and decide if the rules should be altered to cover a larger set of input.

Example Resolved

The Internet bank mentioned in Section 2 was able to determine the cause of the anomalies by inspecting the server logs. The emptying of client accounts was caused by a brute-force attack launched against the bank's authentication mechanism. During the analysis it also became clear that the system lacked validation of business rules.

To rectify the situation the bank integrated an input validator into their web application. The following steps were taken:

1. The authentication mechanism was improved by increasing the password from 4 digits to 6 characters, with the minimum requirements of one lower case letter, one upper case letter, and one digit. All customers were forced to change their passwords using a web form presented at their next log in. The following regular expression rule was created to enforce that new passwords adhered to the new policy:

```
^(?=.*?[a-z])(?=.*?[A-Z])(?=.*?\d)\S{6}$
```

2. A number of business rules were developed to enforce business logic when paying bills. For instance, a rule only allowing numbers greater than zero was specified for the amount field. To prevent customers from entering due dates in the past, the following rules were added:

- (a) A regular expression rule defining `dd.mm.yyyy` as the valid date format:

```
^\d{2}\.\d{2}\.\d{4}$
```

- (b) A rule for checking that the due date is the current date or in the future

Known Uses

A number of worked solutions for input validation exist

Commons Validator [10] is an open-source validation component which originated from the Apache Struts framework. Target objects are JavaBeans. Rules are named *validator actions* and are individual static boolean methods in Java objects. The component enables developers to configure target methods and rules in configuration files.

Stinger [11] is an HTTP Request Validation Engine used in a J2EE environment. The HTTP request is the target object. The engine supports regular expression rules and rules for missing or extra parts in HTTP requests. Developers can specify the rules using the Security Validation Description Language, which is tailored to map the target methods in HTTP requests.

.NET Validation Server Controls [12] are part of Web Forms in the .NET framework, and enable developers to perform input validation on client input in web applications. The target objects are the input server controls, and the framework provides predefined rules such as required field, ranges, and regular expressions. The validation controls are specified as part of the presentation logic.

Consequences

The following benefits can be expected from applying this pattern:

- Input validation can prevent attacks embedded in message content.

- Whitelisting wards off new unknown attacks that fall outside the format dictated by the validators.
- The mechanism enables accounting of attacks by capturing rule violations. The information can later be used to reconstruct attempted security breaks.
- The input validator provides a centralized and manageable mechanism for input validation.
- The same validation rule can be applied to multiple target methods. Input sources that share the same characteristics should reuse the same validation logic.

The following potential liabilities can arise from applying this pattern:

- A client request will carry additional computing overhead, causing delayed server response. The addition of validation logic also causes the server to reach its processing limits sooner.
- The added components introduce new points of failure. Potential bugs can make the application unavailable.
- The new layer of security increases system complexity, which in turn can entail a higher maintenance cost.

See Also

CLIENT INPUT FILTERS [13] discusses the input validation problem in a web context. It points out the inadequacy of client-side checks, and argues that the same checks must be carried out on the server-side as well. The pattern focuses on best practices and issues in implementing web applications.

INTERCEPTING VALIDATOR [14] is a J2EE pattern for validating client input. It is not tied to business logic, and attempts to filter out known attacks early in the request-handling process. In contrast with the INPUT VALIDATOR, the authors recommend using the INTERCEPTING VALIDATOR to also protect against injection attacks.

Useful patterns that can be beneficial in conjunction with the Input Validator pattern include accounting, firewall, and patterns for secure Internet applications [9].

Acknowledgements

Our sincere thanks to Andreas Rüping, who did a great job in shepherding our paper, providing invaluable feedback. Also, we would like to extend our deepest gratitude to the fellow members of our writers workshop group at VikingPLoP 2006: Kristian Elof Sørensen, Aino Vonge Corry, Birgit Zimmerman, Andreas Rüping, Michael Weiss, James O. Coplien, Jayashree Kar, and Rebecca Rikner.

References

- [1] Eurostat. Retrieved May 2006 from http://epp.eurostat.cec.eu.int/portal/page?_pageid=1996,39140985&_dad=portal&_schema=PORTAL&product=_STRIND&root=theme0/strind/innore/ir080&zone=detail.
- [2] ISS E-Security Alert: Form Tampering Vulnerabilities. Retrieved June 2006 from <http://www.cert-rs.tche.br/listas/infoseg/msg00033.html>.
- [3] Open Source Vulnerability Database. Retrieved June 2006 from http://www.osvdb.org/displayvuln.php?osvdb_id=18449.
- [4] J.H. Saltzer and M.D. Schroeder, “The protection of information in computer systems,” in *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [5] J. Yoder and J. Barcalow, “Architectural Patterns for Enabling Application Security,” in *Proceedings of Pattern Languages of Programs (PLoP)*, 1997.
- [6] Data Validation - OWASP. Retrieved June 2006 from http://www.owasp.org/index.php/Data_Validation.
- [7] S.H. Huseby, *Innocent Code—A Security Wake-Up Call for Web Programmers*. John Wiley & Sons, first edition 2004.
- [8] OWASP Top Ten. Retrieved May 2006 from <http://www.owasp.org/documentation/topten.html>.
- [9] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns—Integrating Security and Systems Engineering*. John Wiley & Sons, first edition 2006.
- [10] Commons Validator. Retrieved May 2006 from <http://jakarta.apache.org/commons/validator/>.
- [11] Aspect Security, Stinger HTTP Request Validation Engine. Retrieved April 2006 from <http://www.aspectsecurity.com/stinger/>.
- [12] Validation Server Controls. Retrieved May 2006 from <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconASPNETSyntaxForValidationControls.asp>.
- [13] Client Input Filters. Retrieved May 2006 from <http://www.scrpyt.net/~celer/securitypatterns/>.
- [14] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns—Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, first edition 2005.