

# A Reflection-Based Framework for Content Validation

Lars-Helge Netland  
Department of Informatics  
University of Bergen  
Email: larshn@ii.uib.no

Yngve Espelid  
Department of Informatics  
University of Bergen  
Email: yngvee@ii.uib.no

Khalid A. Mughal  
Department of Informatics  
University of Bergen  
Email: khalid@ii.uib.no

**Abstract**—Attacks embedded in application-level data have become one of the most successful ways to circumvent software security. Skilled hackers capitalize on misplaced trust by concealing their malicious code within a seemingly innocuous stream of application data. In systems that do not perform the most elementary data checks, even unintentional user mistakes may cause a program to behave unexpectedly or crash.

Any distributed software system with potentially untrustworthy sources of input should design and implement a mechanism to inspect application-level data. Such a solution should defend against mischievous attacks, as well as be robust enough to handle user slip-ups. Important steps in creating a successful validation regime include specifying what input to accept, and translating that policy into working code. Once in production, the validation routine must be adaptable in order to accommodate continuously changing requirements.

This paper describes a reflection-based framework for content validation. It separates the inspection of data from the application logic, making it more feasible to construct and maintain a meaningful set of validation rules. The framework is flexible and can be integrated into almost any distributed object-oriented software system. Deployment only requires a basic understanding of XML and expects developers to create a trust model of their own software architecture.

**Keywords:** content validation, distributed systems security, maintainability, reflection, robustness.

## I. INTRODUCTION

A common mistake among developers is misplaced trust in sub-systems. Problems occur when the exchanged bytes deviate from the expected format, which could be orchestrated by an attacker or inadvertently caused by a user. A viable I/O validation strategy combines software security knowledge with risk management analysis, and must be continuously revised to face new threats. A thorough understanding of the environment where the software runs is important in establishing a trust model and identifying forces that can have an impact on the overall security of the system.

Unvalidated input ranks as number one on the Open Web Application Security Project's top ten list [1] over web application vulnerabilities. The problem of handling content is not limited to a Web context, but is equally important in distributed client-server systems running over protocols other than HTTP. The solution described herein is limited to object-oriented software systems, but does not dictate a specific application type or domain. Content validation should be done

prior to processing of data from untrusted sources, and applied throughout the system as needed.

Content validation requirements are likely to change over time, resulting from e.g. application development, feedback from the user community, or anomalies detected during inspection of server logs. It is therefore important to design a solution that can be quickly adjusted to meet changed requirements. The content validator proposed in this paper uses XML and reflection to tackle the dynamic nature of the problem. These two technologies in combination are promising candidates for extending the longevity of software, and can achieve platform-independency [2]. In addition, the suggested solution helps structure the validation process through defining categories of validation rules. These categories encourage developers to think of different types of validation rules.

The work described in this paper builds on [3] and [4]. The former describes a secure communication component that allows developers to focus on application-level programming instead of tedious networking issues, while the latter presents a security pattern for input validation. The framework discussed herein is now available as a SourceForge project called Heimdall [5] under an MIT license.

The rest of this paper is organized as follows: Section II defines content validation and gives an overview of the problem domain; Section III describes other important defense mechanisms that should be considered; Section IV introduces an online banking scenario to show how the framework can be used; Section V explains the inner workings of the content validation framework; Section VI elaborates on the implementation of the online banking example; Section VII discusses related work; Section VIII addresses future work; and Section IX concludes the paper.

## II. MOTIVATION

In a broad sense, I/O can be defined as the *signals* that sub-systems of an information processing system use to communicate with each other. Information received by a functional unit is called *input*, while information sent by the unit is called *output*. In our context, examples of such units include web servers, Internet browsers, databases, and operating systems. The above-mentioned signals are synonymous with data, which can be more appropriately defined as *structured content*. The structure is metainformation, i.e. it describes the format of

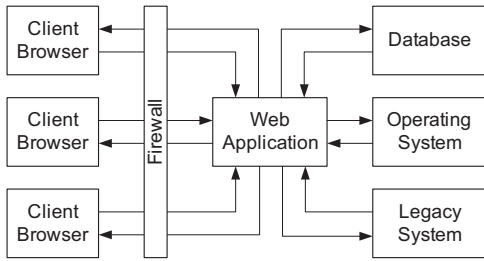


Fig. 1. Architecture

data in a given protocol. The content is the actual information populating the structure. For the purpose of this paper I/O can be defined as structured content exchanged between sub-systems of a larger system.

I/O validation is the process of checking the structure and the contents of data against validation rules. The former means verifying that the information conforms to the protocol format. The latter requires a more careful inspection of the bytes at hand. The difference between structure and content validation can be explained in terms of XML parsing. Checking that an XML message adheres to the Document Type Definition [9] addresses structure validation, while content validation requires analysis of the element and attribute values.

It is also worth noting that classifying information as input or output is a matter of point of view. From the viewpoint of a web server, an HTTP request is input and the corresponding response is output. For an Internet browser, it is reversed. Our communication model does not fit in the producer-consumer paradigm, as many sub-systems take on both roles. As a consequence, the distinction between input and output becomes unimportant. The crucial factor is to establish where the content is to be processed.

Most systems are comprised of several different sub-systems. A typical scenario is shown in Fig. 1. Customers request services from a web server through Internet browsers. A web application receives the initial traffic, and communicates with the back-end sub-systems when serving the clients. A firewall protects against network-level attacks.

A sequence of steps must be completed to create a well-functioning distributed system:

- 1) Identify sub-systems and dataflow. Pinpoint the potential targets for an adversary. Any component in the system that processes input is a possible target.
- 2) Implement the system using defense mechanisms ensuring secure sub-system interaction.
- 3) Specify content validation rules. Define what is to be considered valid content for the system as a whole and

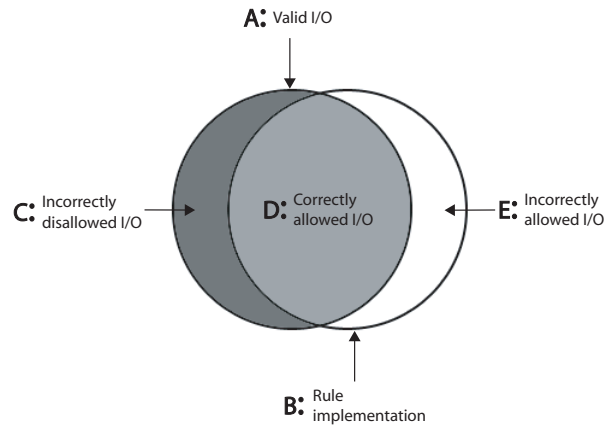


Fig. 2. Implementation mismatch

for each information processing sub-system.

- 4) Implement validation rules. Translate the rule specification into working code and deploy as needed throughout the system.

Step 1 is carried out during design of the system; step 2 is discussed further in Section III; step 3 is exemplified in Section IV by formulating validation rules for bill payment in an online banking scenario; and step 4 is illustrated in Section VI by implementing the rules for bill payment using the framework.

Implementing a perfect I/O validation routine is a very difficult task, mainly due to the rapidly changing environment in which the software operates. As new threats and customer requirements emerge, the solution must be continuously updated to accommodate the changes. Fig. 2 illustrates the relationship between a perfect I/O validation mechanism and an actual implementation. Region A symbolizes the optimal solution, while region B represents a deployed validation routine. The intersection between the two circles, i.e. region D in Fig. 2, embodies I/O that is correctly accepted by the validation logic. Region C represents I/O that is rejected by the current implementation, but should be considered valid according to system policy. This category of I/O does not pose a direct threat in terms of security, but could become a major inconvenience for legitimate users, as input they rightfully consider valid is rejected. Fixing the problem usually involves a user reporting the input that caused a problem to a system operator, who in turn updates the set of rules to accept the incorrectly rejected input. Region E in Fig. 2 represents invalid input that passes undetected through the validation mechanism. Failure to recognize and reject such I/O can have serious security implications. Regions C and E are termed false positives and false negatives, respectively, in the Intrusion Detection Systems community.

Developers should strive to achieve as much overlap as possible between region A and B in Fig. 2. An optimal I/O validation strategy involves maximizing the intersecting region D and minimizing regions C and E. Managing the region E is the key to improving security.

Minimizing region E is not a trivial task. A number of approaches can enhance the quality of I/O validation. Examples include hiring external security expertise for a given application domain, who can help formulate validation rules; investing time and effort into a suitable logging and monitoring system that can reveal new validation needs; and making extensive use of feedback from the user community. Our contribution is a framework that makes it easier to create and maintain a validation regime. At the heart lies the ability to adapt the framework to changing validation requirements. As will be demonstrated later, XML and reflection provide this flexibility.

### III. DEFENSE MECHANISMS

Proper sub-system communication is essential for a well-functioning system. How sub-systems exchange information depend on how they interface. Factors affecting these relationships include the communication techniques offered by the technology and how these mechanisms are made available through APIs. Over time, the communication forms tend to change, often driven by new performance and security requirements. Under these circumstances, developers choose implementation strategies for sub-system communication.

The most dominant server-side sub-system relationship is that of a web application interacting with a back-end database. The first querying technique was string-based, and involved populating SQL commands with client input values. The query string was subsequently submitted and executed in the database. This technique has led to one of the most common venues for client input attacks, namely SQL injection [6]. In short, the attacker alters the querying logic constructed in the web application, by submitting cleverly crafted input, often involving SQL metacharacters. The first defense mechanism against such attacks was to *escape* metacharacters in client input, forcing a literal interpretation in the parser. This is a common approach often used to protect other similar sub-system interactions.

A better solution is to use prepared statements that separate application and querying logic, rendering the context switching attacks impossible. The use of prepared statements removes the need for escaping routines prior to database communication. This evolution highlights the diversity of implementation choices software developers face, and how much proper sub-system communication depends on their knowledge base.

As a contrast to the defense mechanisms just discussed, the main goal for content validation is *not* handling metacharacters or ensuring proper internal communication in a system. The purpose of content validation is to ensure that input conforms to policy rules specified for the system domain. Such rules can define valid structure in credit card numbers, e-mail addresses, etc. Thus, if a rule specifies a valid input set excluding sub-system metacharacters, content validation can have the side-effect of defending against injection attacks as well.

In large software development projects, content validation rules are often implemented by different developers throughout the system code. As a consequence, the larger the system gets,

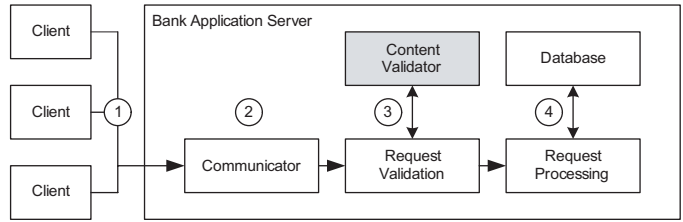


Fig. 3. Online bank architecture

the more difficult it becomes to maintain these rules when requirements change. The process also entails re-compilation and re-deployment of system components.

The following sections demonstrates our generic solution for content validation. In Section IV, we present the architecture of an online banking application and formulate policy rules. The theoretical underpinnings of the framework are presented in Section V, while Section VI shows how to translate the rules specified in Section IV into working code.

### IV. RUNNING EXAMPLE: ONLINE BANKING

Online banking provides a typical scenario in which the content validation framework is useful. Fig. 3 presents the data flow when a client sends a request to the online banking server. The circled numbers indicate the order in which the information propagates, with the lowest number representing the initial point of contact. A scenario for handling a client request can be described by the following steps:

- 1: A bank client sends a request to the bank server.
- 2: A `Communicator` parses the data and creates objects representing the request, based on a communication protocol. These objects are passed to a `Request Validator`.
- 3: The `Request Validator` uses a `Content Validator` to determine which objects in the request are valid. The result of the validation process is a summary of rule violations. Invalid objects are not processed, instead such requests are handled in accordance with policy. E.g., a critical error could cause the bank server to log the incident and terminate the client connection. Valid requests are sent to the `Request Handler`.
- 4: The `Request Handler` determines what to do with the request. Interaction with back-end resources, such as a database, may be necessary to fulfill the request.

The task of developing a secure `Communicator` is discussed in [3].

Describing the validation requirements for an entire Internet banking system falls outside the scope of this paper. We have chosen a scenario for bill payment to show how content validation is set up. The techniques presented are applicable in similar contexts. The example is implemented in Java, and the

Fig. 4. Bill payment web form

enterprise owner is a made-up Spanish bank that needs content validation of domestic payments. The bank uses a web form for bill payments, as shown in Fig. 4. Bank customers are expected to fill in the amount in euros and cents, a payment date, from which account the amount should be withdrawn, and to which account the money should be transferred. Based on application domain expertise, the bank development team can now formulate validation rules for the different fields in Fig. 4:

**Amount** Let  $x$  denote the euro entry, and  $y$  the cent value. Then the following relations must be satisfied:

$$0 \leq x < 1000000$$

$$0 \leq y \leq 99$$

$$x + y > 0,$$

meaning that the amount must be greater than zero and can be up to one million euro.

**Payment date** A valid date has the format  $dd.mm.yyyy$ , where all entries are integers. It should not be possible to backdate bills, i.e. the date must be equal to or after the current date. Also, the date must be a valid Gregorian calendar date.

**From account/To account** Must be 20 digits, in compliance with Spanish domestic account numbers.

## V. A GENERIC VALIDATION MECHANISM

This section provides a top-down walk-through of our content validation approach. First, we present an overview of the main concepts in our work. Next, a class diagram accounts for the core entities and their relationships. After describing the important elements in our solution, a sequence diagram is presented to show how they all interact. In addition, different categories of content validation rules are presented.

### A. Overview

The `Validator` is the driver of the validation process and is the main entity in our validation framework. The validation logic for a `Validator` is configurable in XML. Multiple `Validators` can be created and used throughout the system as needed. Fig. 5 shows how the application can fetch a named `Validator` (1–2) and run the necessary communication in

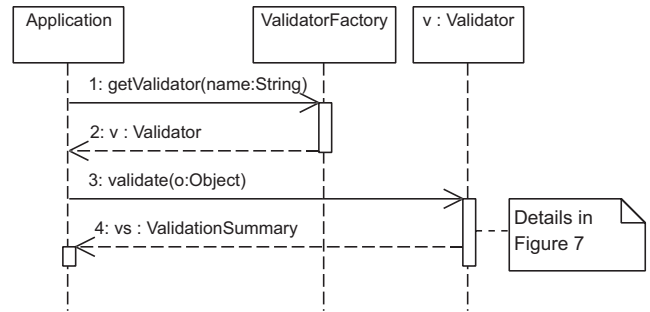


Fig. 5. Using a validator

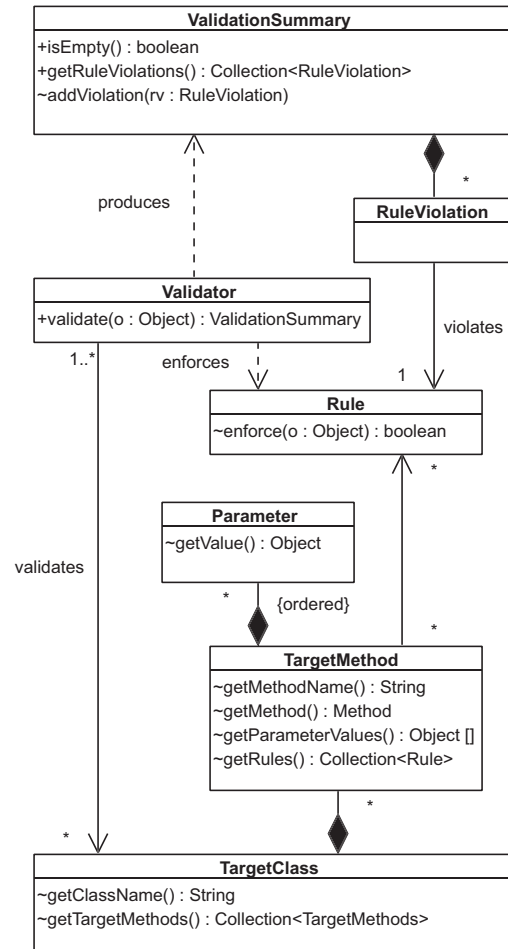


Fig. 6. Validation entities

order to validate a given object. When the application has obtained an object  $o$ , it is submitted for validation to the `Validator` (3). The results from the validation process are summarized in a `ValidationSummary` and returned to the application (4). The application can now take action depending on the information provided in the `ValidationSummary`.

## B. Main entities

Fig. 6 identifies the main entities involved in the validation process. The `Validator` holds a collection of target classes representing objects that must be inspected. Each `TargetClass` holds information on their `TargetMethods` and corresponding method `Parameters`, enabling the `Validator` to retrieve these object properties. In addition, sets of rules define valid return values `TargetMethods` can produce. The `Validator` enforces these rules and produces a `ValidationSummary` containing rule violations. `Rules` and `RuleViolations` represent content validation rules and potential violations of these rules. Return values can be compounded, which necessitates recursive validation.

## C. Validation process

Since the content validation framework is generic and not specific to any application domain, it is unaware of target classes and methods. Class reflection [10] enables the validation framework to determine the class of an object at runtime and invoke its target methods. The configuration of the `Validator` therefore entails specifying class and method names, and defining the corresponding sets of rules. The use of reflection leads to a flexible and dynamic solution for content validation.

Fig. 7 shows the sequence of steps after the target class has been established. The numbers in the left-hand column refer to the steps in the figure.

## D. Rules

Rules constitute the implementation of validation logic. A few common rule categories are already available in the validation framework. For each category, developers can specify initialization parameters, such as range limits, to employ instances of rules in the validation process. The following list describes the different categories:

**Range** Strings, integers etc. should be within certain ranges given by minimum and maximum values.

**Boolean** Boolean values should be either `true` or `false`.

**Required** Values should not be `null`.

**Pattern** Character sequences should adhere to regular expressions.

**Schema** Character sequences should conform to XML schemas.

In scenarios comprising complex validation, the rule categories given above may be insufficient. This limitation is overcome by allowing developers to implement custom validation logic and incorporate it into the framework. By specifying class and method names identifying these custom validation rules, developers are able to use their own validation logic as part of the validation process.

## VI. ONLINE BANKING IMPLEMENTATION

In this section we describe the technical details of the solution implemented for the Spanish bank introduced in Section IV. It should be noted that the validation framework is very flexible, and the following implementation only serves as an example.

First, the validation rules specified in Section IV are translated into XML elements that can be executed by a `Validator`. The rule for payment amount is expressed as follows:

```
<rule name="paymentAmount">
  <range type="java.lang.Double">
    <min>0.01</min>
    <max>999999.99</max>
  </range>
</rule>
```

The Content Validator comes with *built-in* validation logic for ranges. The type must be specified, and the `min` and `max` elements define the valid interval. In this case from 1 cent, inclusive, up to 1 million euros, exclusive.

In most cases, context-specific validation rules must be implemented. Developers can create their own *custom rules* by creating Java classes with methods to enforce these rules. The following XML element for payment date validation shows how to specify custom rules:

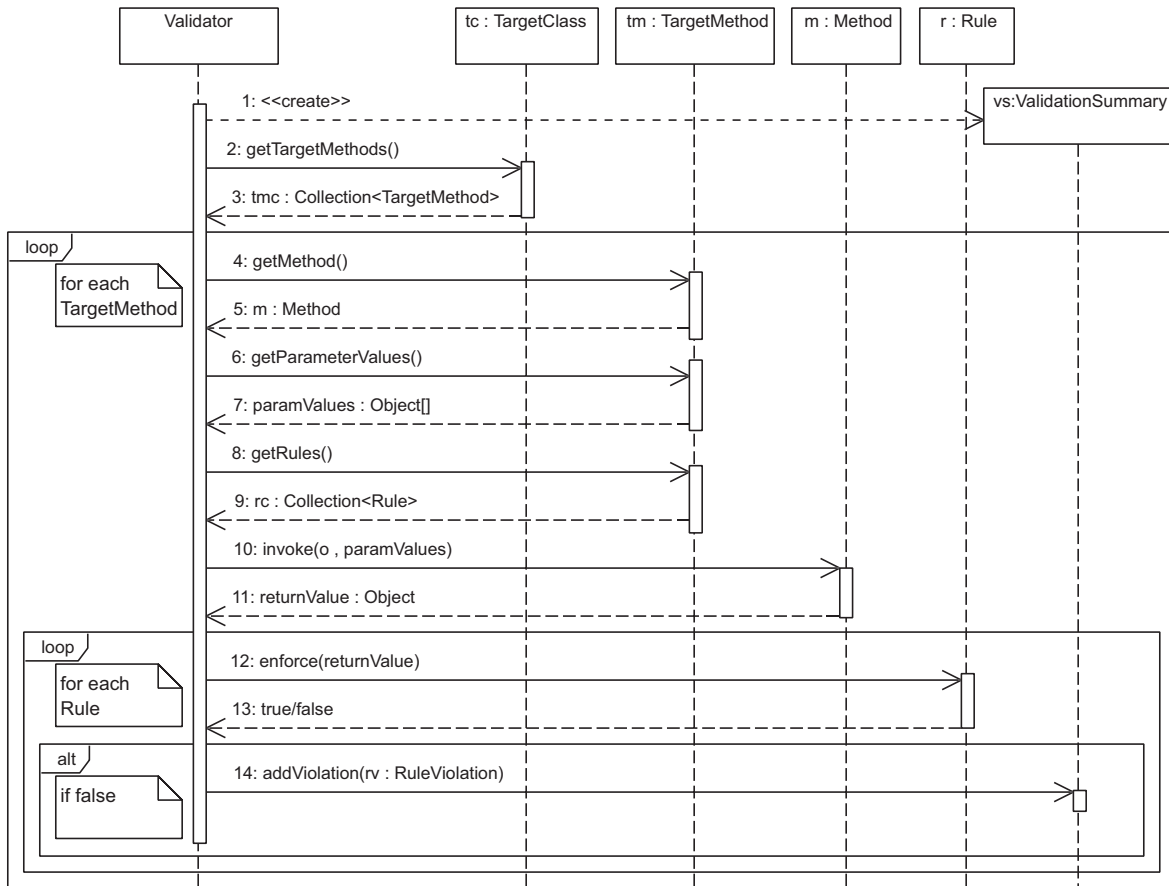
```
<rule name="paymentDate">
  <custom>
    <class>PaymentDateRules</class>
    <method>todayOrInTheFuture</method>
  </custom>
</rule>
```

The `class` element specifies a Java class named `PaymentDateRules`, and the `method` element identifies its static boolean method called `todayOrInTheFuture`. This method validates the date returned by the `Payment` object.

Analogous to range validation, the content validator comes with built-in functionality to specify *regular expression* rules. The following XML segment ensures that an account has exactly 20 digits:

```
<rule name="spanishAccountNumber">
  <regex>^\d{20}$</regex>
</rule>
```

Next, the validation rules need to be associated with target methods. In our simple scenario, a single class holds the payment data. Upon receiving a client request, the `Communicator` shown in Fig. 3 creates a `Payment` object based on the web form input. Fig. 8 shows the getter methods in the `Payment` object that enable the `Validator` to retrieve the payment data. The construction of business objects in the `Communicator` forces a partial syntactical validation of the client data. For instance, a non-numeric amount of euros or cents results in a `NumberFormatException` when



1: The Validator creates a ValidationSummary to hold rule violations.

2-3: The Validator queries the TargetClass for the collection of TargetMethods that should be evaluated on this object.

*The rest of the steps are repeated for all target methods in the collection*

4-5: The Validator retrieves the actual object m which provides access to the class method.

6-7: The parameter values for this method is retrieved by the Validator.

8-9: The set of rules defining a valid return value is retrieved.

10-11: The Validator invokes the method on the object o.

*The rest of the steps are repeated for all rules in the set*

12-13: The return value from step 10-11 is submitted for rule validation.

*The last step is performed if the validation of the rule returns false.*

14: A rule violation is added to the ValidationSummary

Fig. 7. Validation process

Payment
+getTotalAmount() : java.lang.Double
+getPaymentDate() : java.util.GregorianCalendar
+getFromAccount() : java.lang.String
+getToAccount() : java.lang.String

Fig. 8. Object for holding payment data

instantiating the `java.lang.Double` object representing the total amount.

Below, the target methods in the `Payment` class are associated with the validation rules specified earlier:

```
<class name="Payment">
  <method name="getTotalAmount">
    <rulebinding>
      <rule>paymentAmount</rule>
    </rulebinding>
  </method>

  <method name="getPaymentDate">
    <rulebinding>
      <rule>paymentDate</rule>
      <message type="USER">
        Payment date must be the current
        date or a date in the future
      </message>
    </rulebinding>
  </method>

  <method name="getFromAccount">
    <rulebinding>
      <rule>spanishAccountNumber</rule>
    </rulebinding>
  </method>

  <method name="getToAccount">
    <rulebinding>
      <rule>spanishAccountNumber</rule>
    </rulebinding>
  </method>
</class>
```

The `getTotalAmount` method is linked to the `paymentAmount` rule. In addition, different types of messages can be specified in the configuration and used for various purposes such as logging and user response. Above, the string ‘Payment date must be the current date or a date in the future’ can be returned to the client when the system encounters a backdated bill. Also, validation rules can be reused, as illustrated by the `spanishAccountNumber` rule associated with both the `getFromAccount` and `getToAccount` methods.

Multiple validators can be used in a system. Each one of them specifies which classes it validates, as shown below:

```
<validator name="bankValidator">
  <class>Payment</class>
  ...
</validator>
```

## VII. RELATED WORK

Commons Validator [11] is an open-source project originating from the Apache Struts framework that addresses input validation. Their approach centers around a configurable validation engine and a dynamic set of reusable validation methods. The project uses the Java reflection API to create a flexible solution that allows developers to configure and run their own validation logic. The Commons Validator initiative has close ties to JavaBeans, and is primarily concerned with validating fields in Web forms.

Similar solutions include Stinger [12] and Server Validation Controls [13]. The former is an HTTP request validation engine, while the latter provides validation capabilities for Web forms in the .Net framework. Our approach does not require a specific domain or technology, and can be applied in all distributed systems that rely on sub-system communication. E.g. the running example from Sections IV and VI could have been implemented as proprietary bank client software relying on other standards than HTTP and HTML.

Marking and tracking potential harmful data, better known as *tainting*, has been suggested as a solution to the input validation problem. The technique is available in a few programming languages, including Perl [14] and Ruby [15]. In addition, Ngyen-Tuong *et al.* [16] and Haldar *et al.* [17] have developed extensions that enable tainting in PHP and Java, respectively. The taint approach can be fruitful for already deployed applications that were designed without the input validation problem in mind. In such scenarios, tainting can be applied at run-time to identify and track potentially malicious I/O. Any subsequent security sensitive operations involving tainted data are disallowed. This category of input must be *untainted*, i.e. validated, prior to further processing. Developers are trusted to perform meaningful validation. In essence, tainting is an awareness technique that forces people to think about input validation. The approach is a quick fix to cover up for a design flaw, namely a failure to identify untrusted I/O sources. Our solution addresses untrustworthy I/O in the first of the four steps given in Section II, rendering tainting useless, when building new software systems.

## VIII. FUTURE WORK

Reflection incurs performance overhead. To overcome the performance penalty, the framework could replace reflection with code generation, which provides the `Validator` with direct access to target objects. A comparative study between the two techniques would reveal how costly reflection is. We plan to further investigate the relationship between reflection and code generation, and benchmark the results against other similar solutions, such as Commons Validator and Stinger.

## IX. CONCLUSION

Many current distributed software systems lack or implement poor I/O validation. Attackers manipulate the structure or contents of application-level data in order to sidestep defensive measures.

In this paper, we have proposed a content validation strategy to address attacks embedded in message content. The approach includes activities throughout the Software Development Life Cycle. In the design phase it is important to establish trust relationships. All input from sources that are not completely trustworthy should be validated. Implementing the validation logic boils down to configuration, giving developers the opportunity to focus on constructing a viable set of validation rules. Our validation package offers some basic general-purpose rules that can be applied. Developers have the option to add their own rules.

## REFERENCES

- [1] OWASP Top Ten Project. Retrieved November 2006 from [http://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)
- [2] J. Bishop and N. Horspool, "Cross-Platform Development: Software that Lasts," *IEEE Computer*, October 2006, pp. 26–35.
- [3] Y. Espelid, L-H. Netland, K.A. Mughal, and K.J. Hole, "Simplifying Client-Server Application Development with Secure Reusable Components," Proc. International Symposium on Secure Software Engineering (ISSSE), Washington D.C. USA, March 2006.
- [4] L-H. Netland, Y. Espelid, and K.A. Mughal, "Security Pattern for Input Validation," Proc. Viking Pattern Languages of Program (VikingPLoP), Helsingør, Denmark, Sept.–Oct. 2006.
- [5] Heimdall, SourceForge project. Retrieved November 2006 from <http://www.sourceforge.net/projects/heimdall>
- [6] S.H. Huseby, "*Innocent Code—A Security Wake-Up Call for Web Programmers*." John Wiley & Sons, 2004.
- [7] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security—Programming Flaws and How to Fix Them*. McGraw-Hill/Osborne, 2005.
- [8] G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 3, no. 2, 2004, pp. 80–83.
- [9] Extensible Markup Language (XML) 1.0 (Third Edition). Retrieved November 2006 <http://www.w3.org/XML/>
- [10] Reflection. Retrieved November 2006 from <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/index.html>.
- [11] Commons Validator. Retrieved November 2006 from <http://jakarta.apache.org/commons/validator/>.
- [12] SourceForge.net: Stinger HTTP Request Validation Engine. Retrieved November 2006 from <http://sourceforge.net/projects/stinger/>.
- [13] Validation Server Controls. Retrieved November 2006 from <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenreft/html/cpconASPNETSyntaxForValidationControls.asp>.
- [14] Introduction to Perl's Taint Mode. Retrieved November 2006 from <http://www.webreference.com/programming/perl/taint/>.
- [15] Programming Ruby: The Pragmatic Programmer's Guide. Retrieved November 2006 from <http://www.rubycentral.com/book/taint.html>.
- [16] A. Ngyen-Tuong, S. Guarnieri, D. Green, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," Proc. The International Federation for Information Processing Security Conference (IFIP sec), May 2005.
- [17] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," Proc. Annual Computer Security Applications Conference (ACSAC), Dec. 2005.