Tailored Shielding and Bypass Testing of Web Applications

Tejeddine Mouelhi, Yves Le Traon
Security, Reliability and Trust Interdisciplinary Research Center, SnT University of Luxembourg
{tejeddine.mouelhi, yves.letraon}@
uni.lu

Erwan Abgrall
Kereval
Thorigné Fouillard France
Erwan.Abgrall@ Kereval.com

Benoit Baudry
IRISA/INRIA
35042 Rennes France
bbaudry @irisa.fr

Sylvain Gombault
Institut TELECOM ; TELECOM Bretagne ; RSM, 2 rue de la Châtaigneraie CS 17607, 35576 Cesson Sévigné Cedex Université européenne de Bretagne, France
sylvain.gombault@telecom-bretagne.eu

Abstract

User input validation is a technique to counter attacks on web applications. In typical client-server architectures, this validation is performed on the client side. This is inefficient because hackers bypass these checks and directly send malicious data to the server. User input validation thus has to be duplicated from the client-side (HTML pages) to the server-side (PHP or JSP etc.).

We present a black-box approach for shielding and testing web application against bypass attacks. We automatically analyze HTML pages in order to extract all the constraints on user inputs in addition to the JavaScript validation code. Then, we leverage these constraints for an automated synthesis of a shield, a reverse-proxy tool that protects the server side. The originality and main contribution of this paper is to offer a solution specifically tailored to the web application, through a preliminary learning/analysis step. An experimental study on several open-source web-applications evaluates the effectiveness of the protection tool and the different flaws detected by the testing too and the impact of the shield on performance.

1. Introduction

One important property shared by most web applications is the client-server architecture, which roughly divides the application code in two parts. The main part executes on the server, while the client part includes a browser in charge of the interpretation of the HTML and JavaScript code (other components exist like flash, java applets, ActiveX etc.).

In this architecture, the input validation of web application is performed both by the client and server sides. In practice, many validation treatments are under the responsibility of the client. Decentralizing the execution of input validation allows alleviating the load of inputs to be checked by the server-side. Incorrect user inputs are detected by the client-side code (HTML and JavaScript code) and not sent to the server.

This architecture implicitly assumes that the client is expected to check the validity of its inputs before calling the server, while the server is responsible for its outputs. This is a perfect example of design-by-contract [1] that relies on the assumption that both parts are trustable. However, the assumption that the client is trustable is dangerous, as recalled by J. Offutt:[2] "Validating input data on the client is like asking your opponent to hold your shield in a sword fight". It is not possible to trust the execution of the validation on the client side. For this reason, it is highly recommended to duplicate the validation process and perform it at the server side. In addition, input validation is a serious security issue. The SANS TOP 25 [3] reports that one of the main vectors of attacks is input validation. Relying on the client will weaken the input validation. In fact, a malicious user is able to modify the JavaScript code using some plugins (like Firebug or DragonFly). These tools enable the potential attacker to *bypass* the client-side by modifying the HTML and JavaScript code and thus disabling the client-side input validation. Therefore, hackers can bypass the client-side input validation and send malicious requests to the server-side directly. Furthermore, the server cannot detect that client-side input validations have been disabled or hacked. An analysis of

bypass-based attacks has been initially proposed by Offutt et al. [2, 4], demonstrating that the n-tiers architectures may lead to security vulnerabilities, or at least to robustness problems for the server side.

As a basic counter-measure, it is recommended to carefully filter and check user inputs. In this paper, we propose an automated "black-box" process, which either allows:

- Auditing the server-side in order to locate the weaknesses/vulnerabilities (in that case the server-side application code needs to be manually adapted) through systematic bypass testing [2];
- or shielding it by building a reverse proxy security component, called Bypass-Shield that captures the client-side validation constraints, extends them, and enforces them. The shield implements the contracts between client/server as an independent component, making a design-by-contract applicable in the context of web application security and robustness.

The common mechanism we use for both analyses is a semi-automated extraction of client-side validation constraints (HTML and JavaScript).

On one hand, shielding the application involves building an "in-the-middle" component, which is the trustable intermediate that guarantees that contracts are fulfilled by the client (because located on the server-side).

On the other hand, bypass testing involves systematically violating these constraints. Then, requests are built to include some erroneous or malicious data. Finally they are sent to the server and may lead to finding robustness and security problems.

The remainder of this paper is organized as follows. Section 2 presents the background concepts and shows the context of this work. In addition, it describes the scope of this work and presents the limitations and the main differences between our approach and existing approaches. Section 3 explains the overall approach and describes the process. While, Section 4, 5 and 6 detail each of the three processes included in the approach, respectively, the client-side analysis for collecting the constraints, the Bypass-Shield and the automated bypass-testing. Then, Section 7 presents the empirical results. Finally, Section 8 concludes and discusses future work.

2. Context

This section introduces the main concepts used in this paper. It presents the input validation architecture used in web applications and the client-side validation techniques. Afterwards, it details the related work discussing the existing approaches along with their advantages and limitations in both academia and security industry.

2.1. Definitions

Client-side: it includes the part of the web application that is executed by the client browser (Firefox, Internet Explorer etc.). The client-side contains the HTML code, the cookies, the Java applets, and the flash programs etc. that are executed by the client machine.

Server-side: The server side contains the core application code (which is often called business code), the web server which is the framework in which the business code runs. Several technologies of web servers exist (Apache, IIS etc.). The server-side may contain a database.

Input parameter: Web applications may have several input parameters, which are assigned and sent to the server using URLs or forms. For the forms, an input parameter is located in the HTML code. It is a set of tags defining fields to be filled by end users. Then, this data is usually sent to the server-side for processing. The server responds according to the form data and the requested service (store, search, register, delete etc.). For the URLs, input data are hardcoded in the URL and sent back to the server (using JavaScript for example). In this paper, we apply the approach on forms. There are many input vectors for web application that will be taken into account by the shield in future versions.

Pre- and post-condition: The constraint a parameter must satisfy. The client is expected to check the validity of the input before calling the server, while the server is responsible for its outputs.

Input Validation: The process of validating user inputs. It can be performed in the client-side and in the server-side.

Black-box: In this paper, we consider a "black-box" technique any technique that does not require the access to internal information (for instance the application code). Extracting the information (URLs, forms, cookies) that clients can get from the server is thus a black-box technique. This is typically the information a hacker exploits to perform attacks.

Bypass Testing: It is a black-box testing technique which involves bypassing client-side input validation and triggering the server-side input validation (if it exists) [2]. Bypassing is possible either via some browser plugins (like Firebug or Opera Dragonfly) or by automatically generating requests to be sent directly to the server (using Java or C++ for example).

Client-side pre-conditions: They are pre-conditions that are checked by the client. They are expressed by HTML code (like *MaxLength*) or JavaScript functions. These constraints are part of the client-side input-validation process. They enforce limitations and tailored conditions on the user inputs.

Robustness bypass testing: Aiming at challenging the robustness of the web application server-side by directly providing erroneous inputs. These inputs violating the client-side constraints are sent to the server-side. The final goal is to uncover robustness problems and improve the server-side code.

Security bypass testing: Targets the evaluation of server-side security. The data sent in this case represents typical attack vectors (code injection attacks like XSS: cross-site-scripting or SQL Injection). Some predefined attack patterns are injected and sent to the server, which is expected to filter, sanitize or block these malicious data. The final objective is to highlight security issues.

2.2. Client-side validation techniques:

The traditional client-server architecture defines a distributed model which involves two different places where the application code executes.

Erroneous data is detected at an early stage, at client-side and is not sent to the server. Therefore, the code executes on the client machine and the server does not intervene.

Client-side input validation is implemented through two different kinds of code:

- Hardcoded HTML code.
- JavaScript code.

Hardcoded HTML code allows defining a set of predefined constraints. These constraints are implemented by expressing the corresponding tag property. For instance, the max length constraint has to be expressed within the input tag (like `maxlength=20`). Other constraints are expressed by choosing one particular tag. By construction, it constrains the kind of user inputs. Check boxes can only be checked or unchecked. In *radio button group,* only one can be checked.

JavaScript code makes it possible to express more advanced and specific constraints. Using JavaScript code which includes conditions, loops and regular expressions (among other code facilities) it is possible to express any constraints on the user inputs. This JavaScript code can be executed before submitting the form to the server-side. Erroneous inputs are rejected by the JavaScript code and not sent to the server. Then a message is displayed to the end user to indicate the erroneous inputs that should be corrected.

To give the intuition of a typical JavaScript constraint, we present the following JavaScript code that allows checking emails.

```
function checkEmail(myForm) {
if (/^\w+([\.-]?\w+)*@\w+([\.-
]?\w+)*(\.\w{2,3})+$/.test(myForm.emailAddr
.value)) {return true;}
alert("Invalid Email"); return false; }
```

2.3. Scope of the contribution and related work

This paper describes a proxy-firewall for web applications, called bypass-Shield. It checks and blocks invalid user inputs on the server-side. The rules to be checked are automatically inferred from a learning phase (involving parsing the web pages) during which HTML and JavaScript codes are retrieved. The rules can then be manually tuned to offer a tighter control. The learning phase also produces a complete test suite with invalid inputs. These tests can be used to evaluate either the efficiency of the proxy-firewall or the behavior of a web application when invalid inputs are sent.

It is important to note that we do not aim at protecting Ajax-based web apps. There other techniques which target specifically Ajax based web apps (for instance [5]).

The idea of proxy-firewall for web applications is well known and a variety of commercial and free tools already exists. The originality and main contribution of this paper is to offer a solution that is specialized for each application through a preliminary learning phase.

Existing web security techniques help:

1. Auditing/testing vulnerabilities from a black-box perspective (like IBM AppScan [6], HP WebInspect [7], W3AF [8] etc.).
2. Auditing vulnerabilities from a white-box perspective using static analysis of the application code.
3. Protecting/shielding the client side for the server (like BrowserShield [9] etc.).
4. Protect/shield the server side (like ModSecurity [10]) using signature-based techniques.

In this paper, none of these approaches is used to shield the application and test it. The techniques in point 2 and 3 are outside the scope of this paper. White-box auditing aims at cleaning the internal code from potential vulnerabilities before deploying or installing the software. The application code is statically analyzed to detect malware or security breaches. Shielding the client-side is a different task. Several protecting tools can be used to protect clients from security threats. (NoScript for protecting against XSS attacks [11] or web security suites like Norton Internet Security or McAfee etc.).

Black-box audit/testing tools, like the open-source tool W3af [8] are mostly generic tools based on known library of attack patterns that are sent to the server. Most of these tools are specific to one attack pattern and are optimized for one specific web technology. These automated tools cannot replace security experts who can execute more sophisticated attacks based on their knowledge of the web applications. In this paper, we do not focus on generating test cases based on already known patterns but on extracting and violating the specific pre-conditions of the web appli-

cation inputs. Our approach is thus different from these generic tools and tries to assist the task of the security expert who tailors his analysis for a specific web application.

The solution for shielding the server side (point 4) are signature-based in the sense they monitor the inputs that are sent to the server and check if they conform to a specific attack signature (a widely used tool is ModSecurity [10]). The suspected input is sanitized or the request is simply rejected. There are two main drawbacks for these tools. First, they can easily be bypassed using new patterns, for instance by encoding the input to be undetected.

The second main limitation of these tools is that they are not specific to the application. This makes it difficult for these tools to detect attacks that violate the pre-conditions specific to the application, which may lead to the crash of the database (even a max length constraint violation is undetected).

This paper focuses on the second limitation, proposing a test case generation targeting the specific preconditions of a given web application. A list of all these testing and protection tools is maintained by the OWASP community (see [12]).

There are two approaches which are close to our approach [13, 14] and which focus more generally on testing the input validation mechanisms [13] and on bypassing client side validation to discover parameter tampering attacks [14] using a similar approach. However, our technique provides the same testing capabilities and extends it to enable an automated shielding of the web apps against bypass attacks.

Bypassing client-side validation is a well-known security issue and penetration testing has been using client side validation bypass to validate web applications. Offutt et al. formalized the concept of bypass testing [2] and defined its main characteristics. The CyberChair web application (a popular submission and reviewing system used for conferences) served as a feasibility case study to provide initial insights on the efficiency of bypass testing strategy. They tested it using bypass testing strategy and they succeeded to discover serious bugs. For instance, they were able to submit papers without authentication by exploiting bypass testing.

They also proposed an automated tool for bypass testing. They used it to test a simple case study STIS (Small Textual Information System, a web application they built). In their approach, they proposed three different strategies for generating test data. All these strategies target testing the robustness of the web application by sending invalid inputs, set of inputs or by violating the control flow (by breaking expected execution scenarios).

This paper extends and puts more automation in the process of bypass testing to include semi-automated crawling and testing of the security of the web application, distinguishing between security and robustness bypass testing. Testing security involves different test data and a different oracle function than robustness testing. More importantly, the novelty of the proposed approach lies in the Shielding part. The bypass-Shield is constructed using the same artifacts that are used to generate the inputs violating the constraints. By construction, it allows the protection of the web application against these very invalid inputs used to test the servers.

Offutt et al. applied their approach to an industrial case study [15], a web application developed by Avaya Research Labs. They were able to discover 63 failure using 184 test cases. However their approach was not automated and the discovered failures are minor and no security flaw was discovered simply because the bypass tests did not take into account attack patterns.

3. Overview of the approach

From the same initial treatment, the parsing of the web page, the process we propose allows the derivation of a reverse-proxy (Bypass-Shield) and the creation of robustness and security test cases to validate the shield. Figure 1 shows an overview of this process. Two tools have been developed, the Bypass Shield and the Bypass-AutoTest, into the framework of the French ANR DALI project (focusing on application-level intrusion detection).

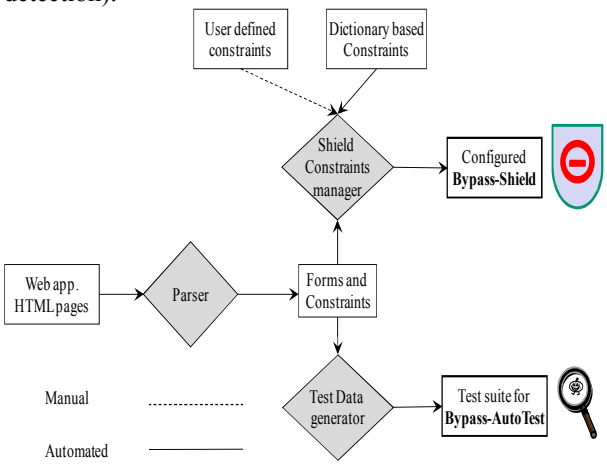

Figure 1 - Pre-condition based Testing and shielding of web applications

The pre-conditions are Boolean expressions that evaluate to true if the value is correct and false when it is not (the input value is violating the pre-condition).The overall process involves three main steps.

The First step involves parsing systematically all pages in order to collect forms along with their respective inputs and pre-conditions. Then these client-side constraints are stored in a file. The result of this step is used in the next two steps. The difficult points and originality of this first step are:

- To exhaustively analyze a website in depth. This means taking into account the login process to access all website pages. In addition to an automated crawler, manual navigation is needed to completely parse the website.
- To deal with JavaScript code used for validating user inputs.

The second step aims at shielding the web application using the initial set of constraints collected at step 1. It results in a reverse-proxy, called Bypass-Shield, which intercepts and checks the inputs from the client as well as server responses. The collected pre-conditions are completed with automated and manual pre-conditions. The automated pre-conditions are based on a dictionary listing a set of constraints to be applied to specific inputs (for instance emails have a specific format). The shield contract manager uses the name of the input to find any available predefined constraint. This task leaves out untreated inputs. The manual addition of constraints completes the automated process by providing a user-friendly tool to add new pre-conditions. The obtained pre-conditions are included in the Bypass-Shield which will intercept user requests and check the validity of user submitted inputs (the tool is available upon request).

The third step involves a test generation process, based on the information collected at step 1. The pre-conditions are used to generate test data for bypass testing. The idea of bypass testing is to generate data which systematically violate the client-side constraints. As a result, we obtain a test tool, Bypass-AutoTest, which allows auditing how the server reacts when receiving every kind of invalid data. Bypass-AutoTest has been implemented first to check that the Shield works as expected and prevents attacks issued by the client-side.

Step2 (Shielding) and step 3 (Auditing through testing) can be used independently or altogether. In the first case, the shield allows protecting the server without modifying the server's code. The advantage is that the security controls are centralized in an independent component, which is responsible for the contracts between client and server. In the other case, the audit allows identifying the server robustness weaknesses and security flaws.

We distinguish between these two kinds of issues. From a pure testing point of view, the oracle, the general interpretation of the results and the impact differ. This means that the intent and the oracle are not the same.

The robustness oracle analyzes the server responses to find error messages (like java stack trace) or unexpected behavior (returning the same page without showing warnings), while the security oracle seeks to find any information or behavior that will harm the security of the application. For instance, the security oracle checks that the server responses does not reveal any critical information that can be used by hackers, or that the server does not behaves in an insecure way.

4. Client-side analysis for pre-conditions identification

This work focuses on bypass-attacks exploiting forms, and does not handle attacks exploiting other attack vectors (like the cookies or HTTP headers). The goal of the HTML analysis is to collect all the user inputs from client-side web pages. User inputs are mainly forms which are filled by the end users. This task is fulfilled using three complementary techniques:

- Automated crawling of the application pages
- Manual navigation in the website to explore all possible scenarios
- Automated navigation using functional test built using testing framework like (HttpUnit [16] or Selenium [17]).

In fact, automated crawling of the web is usually incomplete, and does not reach all the application html pages. Modern web applications use partial page refresh, asynchronous requests and have often just one URL throughout. Visiting all the links will not allow reaching all the HTML pages. In addition, the behavior depends on the client. For these reasons, we complete the automated crawling by two strategies, the manual surfing and the execution of functional tests.

In this section, we show how the automated crawler works and how the bypass shield is used to collect the HTML and finally how we deal with the JavaScript constraints.

4.1. The automated crawler

The crawler allows exploring all the available web pages by visiting all the links. The parser can be configured to use a login and password. This allows going beyond the login web page and exploring the entire web application pages. Furthermore, the parser can be configured to avoid visiting some links that will disconnect the user from the web application. This feature is implemented in a generic way using regular expression to create the set of links to be ignored. For example, all the logout or disconnect links should be avoided. In addition, the parser only visits the pages that are in the base URL. This leads to avoid leaving the web application and parsing other websites/web pages. The crawler runs until all the accessible web

pages are parsed. This crawler allows collecting and storing all the forms along with the associated constraints.

4.2. Manual navigation and Use of functional tests

During this step, testers are asked to run functional tests or navigate manually throughout the web application and to explore all the possible scenarios. During this step the bypass-shield is set in monitoring mode: it collects and analyzes the code to be sent to the client. As shown in Figure 2, the shield intercepts the web pages that are sent to the client and analyzes them in order to collect the forms.

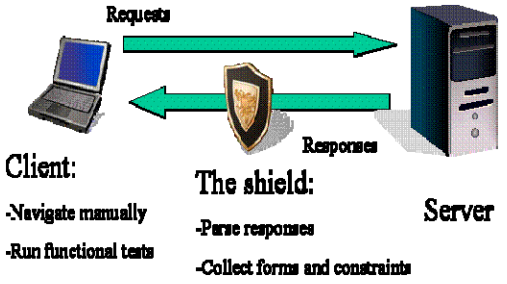

Figure 2 - Collecting pre-conditions using manual navigation and functional tests

The forms and pre-conditions that are collected are stored with the other ones already identified using the crawler. The process of extracting the HTML is common to the crawler and the manual step. Next, we show how the HTML code is analyzed and how the pre-conditions are extracted.

4.3. Collecting HTML constraints

To collect the list of user inputs along with their constraints from the HTML code all web pages should be parsed and analyzed. Each web page is analyzed to locate all the forms. These forms are parsed to collect their inputs. The input may contain some predefined HTML constraints. For instance, we may have max length attribute that defines a pre-condition on the length of an age input.

Table 1 - HTML predefined constraints

Constraint name	Description
FormMethod	Method is either GET or POST and should not be modified.
Disabled	The input is disabled and not sent.
MaxLength	Maximum input size.
MultipleValue	The value should be one of the values set.
ReadOnly	The input is read only and cannot be modified.
RequiredValue	The input value is required and cannot be empty
SingleValue	The input has a single value, Null or that single value

At this stage, only HTML constraints are treated. A separate and parallel process allows dealing with JavaScript code. It will be presented in the next section.

Once all the forms and their inputs are collected, the tool creates a set of objects for each Form and its inputs. We have modeled the types of inputs and the constraints as classes. This approach allows querying forms and inputs and facilitates the test data generation, the construction of the test suite and the configuration of the bypass-shield. Each input is categorized based on its type. Table 1 shows these predefined constraints. For instance, the text input corresponds to InputText object. Each constraint is extracted from the inputs and stored according to its type.

4.4. Interpreting JavaScript

As we have mentioned previously, the JavaScript is not directly parsed. The difficulty of dealing with JavaScript code is due to its grammar which is complex, and this makes the semantic analysis very hard to automate. The solution that we propose involves running the client-side JavaScript validation code itself inside the shied. Instead of inferring the semantic of the JavaScript constraints, we actually run the JavaScript code inside the shield automatically when a form is submitted. We lift the JavaScript code from the client, and then rerun it automatically in the shield.

The main steps of this process involve:

- Locating the JavaScript code implementing constraints on user inputs: the JavaScript validation code is usually triggered just before submitting the form (using for instance the *onsubmit* attribute) or attached to specific text input events (like *onblur* when the user finishes typing and leaves a text input).
- Extracting and storing this code: We should keep a mapping between the JavaScript code and the related form or input.

This process runs in parallel with the extraction of HTML static constraints. We extract for a given web page the JavaScript code related to the input validation. Then we keep a mapping between the JavaScript code and the related form or input.

5. Server-side shield: a shield tool for protecting against bypass attacks

This section presents the bypass-shield and its components. First, we introduce the contract manager tool that allows the addition of constraints to the set that has been generated in previous step. Then, the bypass-shield is presented in details.

5.1. The contracts manager

Table 2 – Examples of constraints

Constraint name	Description
Interval	The value should be within the defined interval
MinLength	Minimum input size
RegEx	The value conforms to the given regular expression
Date	The value has a date format
NumberFormat	The value is numeric
ListOfValues	This value is among a list of values
Required	The input has to be filled
Range	The value is within an interval

The *contracts manager* allows adding new constraints in order to complete the set of constraints extracted from the client's HTML code. Security engineers can add constraints manually through this manager and it also adds new constraints automatically. Table 2 presents some examples of constraints provided by the manager.

The contracts manager automatically injects constraints using a dictionary file, in which the user defines a set of RegEx constraints. These constraints are automatically mapped to input according to their tag names. For instance a tag with the name email will take the following RegEx constraint:

```
^([a-zA-Z0-9_]|\\-|\\.)@(([a-zA-Z0-9_]
|\\-)+\\.)[a-zA-Z]{2,4}$
```

This constraint forces the email addresses to satisfy a specific format. The manager automatically adds this constraint for each email tag, even if the email format was not enforced in the HTML code. This verification is usually added using the JavaScript code. On the basis of the dictionary, the manager can thus partly compensate the fact that we don't analyze JavaScript code. Once the configuration file that is used by the bypass-shield (it is a binary file storing the constraints) is filled with constraints it is fed to bypass-shield which is in charge of protecting the side-side part from bypass-attacks.

5.2. The bypass-shield

As shown in Figure 3, the bypass-shield aims at protecting and serves as a barrier against the attacks. It is installed as a reverse proxy on the server side of the web application. Therefore, all the requests are intercepted by the bypass-shield and checked.

For each request, the bypass-shield performs the following steps:

1. Intercept the request
2. Extract the user inputs and locate the corresponding form that was filled out by the user.
3. Check and validate the input according to the related constraints and run the related JavaScript validation code.
4. Accept the request and send it to the server side application or reject and send an error message to the client.

Only requests containing user inputs are checked. The URL requests are passed to the server. The server is expected to respond by sending back the webpage (the code) of that URL. The user inputs are extracted from the selected requests. In order to locate the corresponding form, the algorithm tries to find among the stored forms (they are stored in a binary file) the one having the same inputs (same number and same name) and the same action URL (the URL to which the inputs are sent). The HTTP request contains all the names of inputs along with their values. The following example illustrates how the algorithm extracts the input names from the request (in this example they are the name, the phone and the zip code). The action URL is simply the request URL without the inputs part.

The request:
http://www.mysite.com/account.php?name=Tim&phone=0234234354&zipcode=75000
The extracted inputs: name, phone and zip code
The action URL:
http://www.mysite.com/account.php

Once the form corresponding to the request is located, the bypass-shield performs the validation of the inputs using the related constraints. All the constraints should be respected. If the inputs do not satisfy the constraints, the request is not forwarded to the server and an error message explaining the problem is sent to the user. In addition, the JavaScript code that is related to the form or one of its inputs is executed on these inputs (using a JavaScript execution engine). The result is a Boolean value (true or false) that means: accept or reject the input data. When all constraints are satisfied and JavaScript validation succeeds, the request is forwarded to the server.

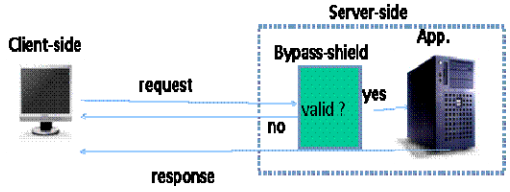

Figure 3 - Overview of the shield

5.3. Impact of enforcing constraints on security

By validating client-side constraints, the shield prevents some code-injection based attacks like SQL injection on numeric fields, by enforcing constraints on numeric fields so it becomes impossible to bypass this constraint and perform any code-injection attack. In addition, it makes it harder for attackers to do long SQL injections when the field length is limited.

By ensuring that the provided parameters are strictly those required, the shields limit IDS evasion techniques like HTTP parameter pollution [18], which is a new kind of attack that involves exploiting parameters in the URL (by duplicating them and injecting attacks).

This kind of enforcement reduces the attack surface of the shielded web application. Using of the shield in front of WebGoat [19] (which is a vulnerable OWASP web application used for teaching security) is a good example to show how the bypass-shield provides extra security, and where it does not. As shown in WebGoat, the developers focus very often the fields that are under user's control (like text fields), and neglect performing input validation on other fields, like check boxes or select lists, which have predefined values. Enforcing constraints on these fields is relatively simple since the expected values are known. By these simple constraints, the shield ensures that the application behaves as expected by the developer and protects against some attacks.

6. Automated bypass testing

This section details the automated generation of bypass testing. The client-side analysis provides useful information on constraints which can be used directly to generate data violating these constraints. On one hand, this data can be used within our bypass-testing too or other security tools like fuzzing tools in order to audit the web application. On the other hand, they could be used for evaluating the bypass-shield.

The data generation process involves three major steps. We start with the automatic generation of malicious test data that violate the client-side constraints. Then, we build complete requests that include the malicious data and for which all other tags contain valid data. These requests are sent to the server-side. The last step involves the analysis of the server responses and the automated classification of the results, in order to facilitate their interpretation by the testers.

6.1. The generation of malicious test data

The initial step involves generating test data that bypass the client-side constraints. For each constraint, we have created a data generator in our Bypass-AutoTest tool. This data generator is in charge of creating the data violating a specific constraint. The following example illustrates the approach.

For example, when the input is a phone number with a maximum length limit (10 characters), the data generator takes this input and its constraint and generates a random string with a length exceeding the required max length by 10. The interval of violation can be defined by the user (10 is the default value).

Table 3 shows some examples of constraints and the generated data.

Table 3 - The way constraints are violated

Constraint	Violation
FormMethod	Use another form method
Disabled	Make it enabled and generate a random string
MaxLength	Generate data exceeding MaxLength
MultipleValue	Generate a different random value
RegEx	Create a value not conform to RegEx
Date	Create a random value that is not a date
NumberFormat	Generate a string with alphabetic characters
ListOfValues	Generate a value not in the list of values
Range	Generate a value outside that range

6.2. Construction and execution of bypass tests

This step requires the construction of suitable requests from the set of test input generated in previous step. For the request to be valid, all the form tags must be filled. In fact, each test request contains only one unique malicious input; all other inputs are valid with respect to the constraints.

The fact that there is only one single malicious data in each test request allows avoiding any side-effects due to the server-side rejecting the request. Also, if the test fails, revealing the lack of input validation or a serious security flaw, the fact that each request contains only one maliciously input facilitates the localization of the source of this problem.

To generate these requests, the malicious and genuine data are combined to fill the forms. Then, the requests are sent to the server side to be processed. Afterwards, the server responds and all these responses are stored in order to be interpreted and classified.

7. Experiments and results

This section presents an evaluation of both the *protection technique* using three case studies, which are JForum, Insecure and DVWA (Damn Vulnerable Web application) and the *bypass testing technique* using four case studies (JForum, Roller, PhpBB and MyReview). JForum and PhpBB are widely used web applications that help creating forums. Roller allows creating customized blogs while MyReview is a conference management tool. Finally, Insecure and DVWA are vulnerable web applications used to demonstrate web attacks and to evaluate the protection techniques.

This section presents and discusses several results. Firstly, we evaluate the number of forms that are automatically parsed using our tool. We calculate the number of forms that are parsed through manual navigation in order to estimate the effort needed to parse all the forms. Secondly, we calculate the number of vulnerabilities that are stopped by the bypass-shield. Thirdly, we evaluated the testing tool by applying it to

four popular web applications including JForum. The idea is to evaluate the ability of the bypass testing tool to discover new vulnerabilities or robustness issues in web applications. Finally, we estimate the overhead due to the use of the shield by calculating the additional latency (which is almost constant for a given configuration). In fact, duplicating constraints in an 'in-the-middle' shield may create an overhead. The results provide evidence of the fact that the shield is a lightweight solution.

7.1. Parsing results

We applied our parser to automatically get all the forms. Table 4 shows the number of forms.

The results show that the number of forms that are automatically discovered is much bigger than those parsed using manual navigation (80% for Insecure and 65% for JForum and 100% for DVWA). The results show that the automated parser helps getting most of the forms.

Table 4 - Automated vs. manual parsing

	Insecure	JForum	DVWA
Automated parsing	12	22	23
Manual navigation	3	13	0
Total	15	34	13

7.2. Bypass shielding results

This section presents an evaluation of the effectiveness of the shield in stopping attacks. Using classical penetration testing techniques and using the WA3F tool (the third co-author is a security expert), we were able to find 9 exploitable vulnerabilities in JForum. For the other two web applications, the vulnerabilities are well known and documented since they are vulnerable by construction. Table 5 shows the overall number of vulnerabilities for each application and the number of vulnerabilities that were mitigated thanks to the bypass-shield using automatically retrieved constraints and manually added constraints. Most of these vulnerabilities are related to weak server-side validation of user input, which enable performing attacks like: SQL Injection, XSS and DOS: Denial of Service, which try to send a huge amount of data to the server to make it crash. By duplicating client side constraints, the shield allows mitigating these vulnerabilities. By restricting the content and length of fields like name or phone number, the shield allows blocking attacks (SQL injection or DOS attacks).

Table 5 - Vulnerabilities mitigated by the shield

	Insecure	**JForum**	**DVWA**
Without shield	15	9	5
With shield	3	5	3

The remaining vulnerabilities that are not mitigated by the shield are related to XSS which exploit text fields. Even with specific constraints the shield was not able to stop these XSS attacks.

7.3. Bypass testing results

The bypass testing results are shown in Table 6. Using the bypass testing tool, we were not able to discover any serious issue in both phpBB3 and Roller. However, we were able to find some robustness problems, especially in the JForum application.

In fact, the tests provoked 353 failures related to three kinds of Java exceptions:

- Null Pointer Exception: Use of a null variable. It occurs when null inputs are sent to server.
- Class Cast Exception: incompatible class type cast. When unexpected input is sent to server (an input that is not is a predefined list).
- Number Format exception: The server tried to convert a string into an integer. It occurs when non numeric values are sent instead of numbers.

Table 6 – Bypass testing results

App.	**#Failures**	**#SQL failures**	**#Null Response**	**Responses codes**
JForum	Java: 353	1	0	[302, 404]
phpBB3	0	0	183	-
Myreview	0	1	650	-
Roller	0	0	0	[405, 500]

These failures are due to bugs in the input validation code located in the server-side. The server did not check correctly the user inputs. In addition, for two web applications (phpBB3 and MyReview), we received 'Null responses'. The server returned empty responses. Furthermore, according to the response code, there were three kinds of responses:

- Response 404: The requested page is not found. This occurred when hidden values were modified. The server uses them to reach certain kinds of pages. When the hidden value is not correct, this leads to the response 404.
- Response 405: The method is not allowed (using GET method when submitting a form instead of POST).
- Response 500: Internal server error.

We found two SQL flaws in MyReview and in JForum. The JForum one originated from a form used to submit new posts in the forum where the input subject length is not checked by the server side. When a long string is sent to the server an SQL Exception occurs and the SQL query is exposed to users. This vulnerability was discovered manually by our security expert when he performed penetration testing on JForum.

7.4. Performance results

To measure the overhead due to the bypass shield, we generated 50 instances of each form in JForum and run it with and without the shield in order to record differences in the execution time. We repeated this

process ten times and calculated the average of execution times. We calculated a constant overhead value of 3 ms (with our server configuration).

8. Conclusion and Future Work

This paper presented a new approach that aims at automating the shielding of web-applications against bypass attacks. The novelty of the approach resides in the analysis of the HTML code to extract constraints on the user inputs in addition to the JavaScript validation code. These inputs are used to build a shield that executes as a reverse proxy to enforce these constraints. This tool suite will be extended to cope with other security issues. A new study is in progress that will use the shield to protect against attacks other that bypass attacks. The bypass-shield and its client constraints and inputs that are collected constitute an interesting platform to implement new kinds of protection strategies.

Another research direction would be to apply this approach to rich internet application (like Ajax or Flex) where client-server communication is asynchronous which makes the analysis very difficult. Automatically analyzing the traffic between the client and the server may help coping with this technology.

9. References

1. J.M. Jézéquel, Michel Train, and Christine Mingins, *Design Patterns and Contracts*. 1999: Addison-Wesley. 348.
2. J. Offut, Y. Wu, X. Du, and H. Huang. *Bypass testing of Web applications*. in *15th International Symposium on Software Reliability Engineering ISSRE 2004* 2004.
3. *CWE/SANS TOP 25 Most Dangerous Programming Errors*. Available from: http://www.sans.org/top25errors/.
4. J. Offutt, Q. Wang, and J. Ordille. *An Industrial Case Study of Bypass Testing on Web Applications*. in *Software Testing, Verification, and Validation, 2008 International Conference on*. 2008.
5. K. Vikram, Abhishek Prateek, and Benjamin Livshits, *Ripley: automatically securing web 2.0 applications through replicated execution*, in *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, ACM: Chicago, Illinois, USA. p. 173-186.
6. *IBM AppScan* http://www-01.ibm.com/software/awdtools/appscan/.
7. *HP WebInspect* *https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200^9570_4000_100__*.
8. *W3AF:* http://w3af.sourceforge.net/.
9. C. Reis, J. Dunagan, J. Wang Helen, O. Dubrovsky and S. Esmeir, *BrowserShield: vulnerability-driven filtering of dynamic HTML*, in *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, USENIX Association: Seattle, Washington.
10. *ModSecurity* http://www.modsecurity.org/.
11. *NoScript* http://noscript.net/.
12. *OWASP security and testing tools:* http://www.owasp.org/index.php/Phoenix/Tools.
13. H. Liu and H. Beng Kuan Tan, *Automated verification and test case generation for input validation*, in *Proceedings of the 2006 international workshop on Automation of software test*. 2006, ACM: Shanghai, China. p. 29-35.
14. P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, *NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications*, in *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, ACM: Chicago, Illinois, USA. p. 607-618.
15. A. Bertolino, A. Polini, P. Inverardi, H. Muccini, and Proc. (Extended abstract) " , , June 28- 1 July 2004, Pag. 124-125. *Towards Anti-Model-based testing*. in *International Conference of Dependable Systems and Networks DSN 2004*. 2004. Florence.
16. A. Tappenden, P. Beatty, and J. Miller. *Agile Security Testing of Web-Based Systems via HTTPUnit*. in *Agile Development Conference*. 2005.
17. *Selenium:* http://seleniumhq.org/.
18. L. Mike Ter and V. N. Venkatakrishnan, *Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers*, in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. 2009, IEEE Computer Society.
19. P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, *SWAP: Mitigating XSS attacks using a reverse proxy*, in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*. 2009, IEEE Computer Society.