Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis

Theodoor Scholte
SAP Research, Sophia Antipolis, France
theodoor.scholte@sap.com

William Robertson
Northeastern University, Boston, USA
wkr@ccs.neu.edu

Davide Balzarotti
Institut Eurecom, Sophia Antipolis, France
davide.balzarotti@eurecom.fr

Engin Kirda
Northeastern University, Boston, USA
ek@ccs.neu.edu

***Abstract*—Web applications have become an integral part of the daily lives of millions of users. Unfortunately, web applications are also frequently targeted by attackers, and criticial vulnerabilities such as XSS and SQL injection are still common. As a consequence, much effort in the past decade has been spent on mitigating web application vulnerabilities.**

Current techniques focus mainly on sanitization: either on automated sanitization, the detection of missing sanitizers, the correctness of sanitizers, or the correct placement of sanitizers. However, these techniques are either not able to prevent new forms of input validation vulnerabilities such as HTTP Parameter Pollution, come with large runtime overhead, lack precision, or require significant modifications to the client and/or server infrastructure.

In this paper, we present IPAAS, a novel technique for preventing the exploitation of XSS and SQL injection vulnerabilities based on automated data type detection of input parameters. IPAAS automatically and transparently augments otherwise insecure web application development environments with input validators that result in significant and tangible security improvements for real systems. We implemented IPAAS for PHP and evaluated it on five real-world web applications with known XSS and SQL injection vulnerabilities. Our evaluation demonstrates that IPAAS would have prevented 83% of SQL injection vulnerabilities and 65% of XSS vulnerabilities while incurring no developer burden.

I. INTRODUCTION

Web applications have become attractive targets for attackers due to the large degree of authority they possess, their significant user populations, and the prevalence of vulnerabilities they contain. Among the classes of vulnerabilities exhibited by web applications, XSS and SQL injection remain among the most serious threats to web application security. As such, much attention in the security research community has focused on removing or mitigating the effect of these vulnerabilities [2], [12], [18], [35].

XSS and SQL injection vulnerabilities both manifest at a fundamental level as a failure to preserve the integrity of HTML documents and SQL queries, respectively, in the presence of untrusted input to the web application. In the former case, an XSS vulnerability allows an attacker to inject dangerous HTML elements, typically including malicious client-side code that executes in the security context of a trusted web origin. In the latter case, a SQL injection injection vulnerability allows an attacker to modify an existing database query — or, in some cases, to inject a completely new one — in such a way that violates the web application's desired data integrity or confidentiality policies.

One particularly promising approach to preventing the exploitation of these vulnerabilities is robust, automated *sanitization* of untrusted input. In this approach, filters, or *sanitizers*, are automatically applied to user data such that dangerous constructs cannot be injected into HTML documents or SQL queries. Automated protection against these vulnerabilities is highly desirable due to the well-known difficulty in manually achieving complete and correct sanitizer coverage.

Output sanitization: A particularly promising approach in this vein is automated *output sanitization*, where sanitizers are automatically applied to data computed from untrusted data immediately prior to its use in document or query construction [23], [27], [36]. Output sanitization that is automated, context-aware, and robust with respect to real browsers and databases is an extremely attractive solution to preventing XSS and SQL injection attacks. This is because it provides a high degree of assurance that the protection system's view of untrusted data used to compute documents and queries is identical to the real system's view. That is, if an output sanitizer decides that a value computed from untrusted data is safe, then it is almost certainly the case that that data is actually safe to render to the user or submit to the database.

Unfortunately, output sanitization is not a panacea. In particular, in order to achieve correctness and complete coverage of all locations where untrusted data is used to build HTML documents and SQL queries, it is necessary to construct an abstract representation of these objects in order to track output contexts. This generally requires the direct specification of documents and queries in a domain-specific

language [23], [27], or else the use of a language amenable to precise static analysis. While new web applications have the option of using a secure-by-construction development framework or templating language, legacy web applications do not have this luxury. Furthermore, many web developers continue to use insecure languages and frameworks for new applications.

Input validation: In contrast to output sanitization, another approach for preventing XSS and SQL injection injection vulnerabilities is the use of *input validation*. Input validation involves checking the inputs to the web application against a specification of legitimate values (e.g., a certain parameter should be an integer, or an email address, or a URL). Input validation is more general than output sanitization in the sense that input validation has the broader goal of program correctness rather than preventing specific classes of attacks. However, input validation provides less assurance that vulnerabilities will be prevented, since it relies on check routines to validate untrusted input, but it may still fail to identify the input as being malicious. In addition to that, untrusted data can also undergo potentially arbitrary transformations, as part of application processing prior to being output into a document or query, making input validation ineffective.

We note, however, that despite these drawbacks, input validation has significant benefits as well. First, even though input validation is not necessarily focused on enforcing security constraints, rigorous application of robust input validators has been shown to be remarkably effective at preventing XSS and SQL injection attacks in real, vulnerable web applications [30], [31]. For instance, we have demonstrated that robust input validation would have been able to prevent the majority of XSS and SQL injection attacks against a large corpus of known vulnerable web applications.

Second, it is comparatively simple to achieve complete coverage of untrusted input to web applications as opposed to the case of output sanitization. Web application inputs can be enumerated given *a priori* knowledge of the language and development framework, whereas context-aware output sanitization imposes strict language requirements that often conflict with developer preferences. Consequently, input validation can be applied even when insecure legacy languages and frameworks are used.

IPAAS: In this work, we present IPAAS (**I**nput **PA**rameter **A**nalysis **S**ystem). IPAAS transparently integrates robust, automated input parameter validation into the web application development environment. Specifically, IPAAS automatically (i) extracts the parameters for a web application; (ii) learns types for each parameter by applying a combination of machine learning over training data and a simple static analysis of the application; and (iii) automatically applies robust validators for each parameter to the web application with respect to the inferred types.

We have implemented IPAAS for PHP in the form of an OWASP WebScarab extension to extract and learn parameter types, and a runtime PHP rewriting component to enforce proper validation of parameter values. We evaluated our system over five real-world PHP-based applications containing numerous XSS and SQL injection vulnerabilities, and demonstrate that IPAAS would prevent 83% of known SQL injection attacks and 65% of known XSS attacks against the set of test applications.

Unfortunately, due to the inherent drawbacks of input validation, IPAAS is not able to protect against all kind of XSS and SQL injection attacks. However, our experiments show that IPAAS is a simple and effective solution that can greatly improve the security of web applications. Our technique automatically and transparently applies input validators during the development phase of a web applications. Therefore, IPAAS helps developers that are unaware of web application security issues to write more secure applications.

Contributions: To summarize, in this work, we make the following contributions.

- We identify automated input validation as an effective alternative to output sanitization for preventing XSS and SQL injection vulnerabilities in legacy applications, or where developers choose to use insecure legacy languages and frameworks.
- We implement the IPAAS approach for transparently learning types for web application parameters, and automatically applying robust validators for these parameters at runtime.
- We evaluate our implementation for PHP, and demonstrate its ability to prevent the exploitation of 65% of XSS vulnerabilities and 83% of SQL injection vulnerabilities with a low false positive rate over five real-world PHP applications.

The remainder of the paper is structured as follows. In §II, we discuss our motivations for studying input validation as a prevention mechanism for XSS and SQL injection vulnerabilities. §III presents the IPAAS approach to automatically applying robust input validation to web applications. §IV evaluates our implementation of IPAAS for PHP over a test set of vulnerable PHP applications. We present related work in §V. In §VI, we conclude and discuss potential future work.

II. Background

Input validation and sanitization are related techniques for helping to ensure correct web application behavior. However, while these techniques are related, they are nevertheless distinct concepts. Sanitization — in particular, output sanitization — is widely acknowledged as the preferred mechanism for preventing the exploitation of XSS vulnerabilities. In this section, we highlight the advantages of input validation, and thereby motivate the approach we present in following sections.

Input validation is fundamentally the process of ensuring that program input respects a specification of legitimate

```
POST /payment/submit HTTP/1.1
Host: shop.example.com
Cookie: SESSION=cbb8587c63971b8e
[...]

cc=1234567812345678&month=8&year=2012&
save=false&token=006bf047a6c97356
```

Figure 1. HTTP POST request containing several examples of untrusted program input.

values. Any program that accepts untrusted input should incorporate some form of input validation procedures, or *input validators*, to ensure that the values it computes are sensible. The validation should be performed prior to executing the main logic of the program, and can vary greatly in complexity. At one end of the spectrum, programs can apply what we term *implicit validation* due to, for instance, typecasting of inputs from strings to integers in a statically-typed language. On the other hand, programs can apply *explicit validation* procedures that check whether program input satisfies complex structural specifications, such as the Luhn check for credit card numbers.

In the context of web applications, input validation should be applied to all untrusted input; this includes input vectors such as HTTP request query strings, POST bodies, database queries, XHR calls, and HTML5 postMessage invocations. As an example, consider the POST request shown in Figure 1. The request contains several parameters, including: `cc`, a credit card number; `month`, a numeric month; `year`, a numeric year; `save`, a flag indicating whether the payment information should be persisted for future use; `token`, a CSRF nonce; and `SESSION`, a session identifier. Each of these request parameters requires a different type of input validation. For example, the credit card number should contain certain characters and pass a Luhn check. The month should be an integer between 1 and 12. The year should be an integer value representing a year in the near future. Finally, the save parameter should contain a boolean value (e.g., “0”, “1”, “true”, “false”, or “yes”, “no”).

Input validation is concerned with a broader goal of program correctness, while sanitization focuses on the specific goal of removing dangerous constructs from values computed using untrusted data. Sanitation procedures, or *sanitizers*, focus on enforcing a particular security policy, such as preventing the injection of malicious JavaScript code into an HTML document. While rigorous input validation can provide a security benefit as a side-effect, sanitizers should provide a strong assurance of protection against particular classes of attacks.

Sanitizers have traditionally been applied throughout the web application processing lifecycle, but automated *output sanitization* has come to be recognized as the most attractive form of the technique. Sanitizing untrusted data immediately prior to its use is highly desirable because it provides a high degree of assurance that the protection system’s view of untrusted data is identical to the real system’s view. Input validation in isolation, on the other hand, cannot guarantee that an input it considers safe will not be transformed during subsequent processing into a dangerous value.

```
<div class="msg">
    <h1 style="${msg.style}">${msg.title}</h1>
    <p>${msg.body}</p>
</div>
```

Figure 2. HTML fragment output sanitization example.

As an example of output sanitization, consider the web template fragment shown in Figure 2. Here, untrusted input is interpolated as both child nodes of the `h1` and `p` DOM elements, as well as in the `style` attribute of the `h1` element. At a minimum, a robust output sanitizer should ensure that dangerous characters such as ‘`<`’ and ‘`&`’ should not appear un-escaped in the values to be interpolated, though more complex element white-listing policies could also be applied. Additionally, the output sanitizer should be *context-aware*; for instance, it should automatically recognize that ‘`"`’ characters should also be encoded prior to interpolating untrusted data into an element attribute. The output sanitizer described here would be able to prevent attacks that might bypass input validation. For instance, an input verified to be valid might nevertheless be concatenated with dangerous characters during processing before being interpolated into a document.

III. IPAAS

In this section, we present IPAAS (**I**nput **PA**rameter **A**nalysis **S**ystem), an approach to securing web applications against XSS and SQL injection attacks using input validation. The key insight behind IPAAS is to automatically and transparently augment otherwise insecure web application development environments with input validators that result in significant and tangible security improvements for real systems.

IPAAS can be decomposed into three phases: (i) parameter extraction, (ii) type learning, and (iii) runtime enforcement. An architectural overview of IPAAS is shown in Figure 3. In the remainder of this section, we describe each of these phases in detail.

A. Parameter Extraction

The first phase is essentially a data collection step. Here, a proxy server intercepts HTTP messages exchanged between a web client and the application during testing. For each request, all observed parameters are parsed into key-value pairs, associated with the requested resource, and stored in

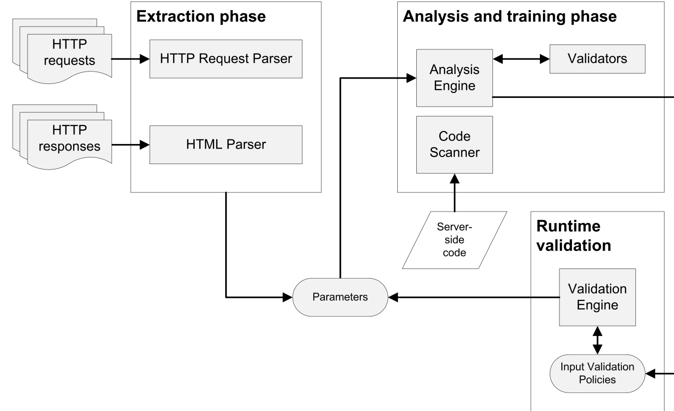

Figure 3. The IPAAS architecture. A proxy server intercepts HTTP messages generated during application testing. Input parameters are classified during an analysis phase according to one of a set of possible types. After sufficient data has been observed, IPAAS derives an input validation policy based on the types learned for each application input parameter. This policy is automatically enforced at runtime by rewriting the application.

Type	Validator
`boolean`	`(0|1)|(true|false)|(yes|no)`
`integer`	`(+|-)?[0-9]+`
`float`	`(+|-)?[0-9]+(\+.[0-9]+)?`
`URL`	*RFC 2396, RFC 2732*
`token`	*static set of string literals*
`word`	[0-9a-zA-Z@_-:]+
`words`	[0-9a-zA-Z@_-: \r\n\t]+
`free-text`	*none*

Table I
IPAAS TYPES AND THEIR VALIDATORS.

a database. Each response containing an HTML document is processed by an HTML parser that extracts links and forms that have targets associated with the application under test. For each link containing a query string, key-value pairs are extracted similarly to the case of requests. For each form, all input elements are extracted. In addition, those input elements that specify a set of possible values (e.g., `select` elements) are traversed to collect those values.

B. Parameter Analysis

The goal of the second phase is to label each parameter extracted during the first phase with a data type based on the values observed for that parameter. The labeling process is performed by applying a set of validators to the test inputs.

1) Validators: Validators are functions that check whether a value meets a particular set of constraints. In this phase, IPAAS applies a set of validators, each of which checks that an input belongs to one of a set of types. The set of types and regular expressions describing legitimate values are shown in Table I. In addition to the types enumerated in Table I, IPAAS recognizes lists of each of these types.

2) Analysis Engine: IPAAS determines the type of a parameter in two sub-phases. In the first, types are learned based on values that have been recorded for each parameter. In the second, the learned types are augmented using values extracted from HTML documents.

Learning: In the first sub-phase, the analysis begins by retrieving all the resource paths that were visited during application testing. For each path, the algorithm retrieves the unique set of parameters and the complete set of values for each of those parameters observed during the extraction phase. Each parameter is assigned an integer score vector of length equal to the number of possible validators.

The actual type learning phase beings by passing each value of a given parameter to every possible type validator. If a validator accepts a value, the corresponding entry in that parameter's score vector is incremented by one. In the case that no validator accepts a value, then the analysis engine assigns the `free-text` type to the parameter and stops processing its values.

After all values for a parameter have been processed, the score vector is used to select a type and, therefore, a validator. Specifically, the type with the highest score in the vector is selected. If there is a tie, then the most restrictive type is assigned; this corresponds to the ordering given in Table I.

The second sub-phase uses the information extracted from HTML documents. First, a check is performed to determine whether the parameter is associated with an HTML `textarea` element. If so, the parameter is immediately assigned the `free-text` type. Otherwise, the algorithm checks whether the parameter corresponds to an `input` element that is one of a checkbox, radiobutton, or `select` list. In this case, the observed set of possible values are assigned to the parameter. Moreover, if the associated element is a checkbox, a multi-valued `select`, or the name of the parameter ends with the string `[]`, the parameter is flagged as a list.

The analysis engine then derives input validation policies for each parameter. For each resource, the path is linked to

the physical location of the corresponding application source file. Then, the resource parameters are grouped by input type (e.g., query string, request body, cookie) and serialized as part of an input validation policy. Finally, the policy is written to disk.

Static Analysis: The learning sub-phases described above can be augmented by static analysis. In particular, IPAAS can use a simple static analysis to find parameters and application resources that were missed during the learning phase due to insufficient training data. This analysis is, of course, specific to a particular language and framework. We describe our prototype implementation of the static analysis component in Section III-D.

C. Runtime Enforcement

The result of the first two phases is a set of input validation policies for each input parameter to the web application under test. The third phase occurs during deployment. At runtime, IPAAS intercepts incoming requests and checks each request against the validation policy for that resource's parameters. If a parameter value contained in a request does not meet the constraints specified by the policy, then IPAAS drops the request. Otherwise, the application continues execution.

A request may contain parameters that were not observed during the previous phases, either in the learning sub-phases or static analysis. In this case, there are two possible options. First, the request can simply be dropped. This is a conservative approach that might, on the other hand, lead to program misbehavior. Alternatively, the request can be accepted and the new parameter marked as valid. This fact could be used in a subsequent learning phase to refresh the application's input validation policies.

D. Prototype Implementation

Parameter extraction: We have implemented a prototype of the IPAAS approach for PHP. Parameter extraction is performed by a custom OWASP WebScarab extension, and HTML parsing performed by jsoup. WebScarab is a client-side interceptor proxy, but this implementation choice is of course not a restriction of IPAAS. The extractor could have easily been implemented as a server-side component as well, for instance as an Apache filter.

Type learning: The parameter analyzer was developed as a collection of plugins for Eclipse and makes use of standard APIs exposed by the platform, including JFace and SWT. The Java DOM API was used to read and write the XML-based input validation policy files.

Static analyzer: We implemented a simple PHP static analyzer using the Eclipse PHP Development Tools (PDT). The analyzer scans PHP source code to extract the set of possible input parameters. There are many ways in which a PHP script can access input parameters. In simple PHP applications, the value of an input parameter is retrieved by accessing one of the following global arrays: `$_GET`, `$_POST`, `$_COOKIE`, or `$_REQUEST`. However, in more complex applications, these global arrays are wrapped by special library functions that are specific to each web application.

In order to collect input parameters for PHP, our static analyzer performs pattern matching against source code and records the name of input parameters. The location of the name of an input parameter can be specified in a pattern. A pattern can be specified as a piece of PHP code and is attached to one or more input vectors (e.g., `$_GET`). For example, the pattern `optional_param('$', '*')` specifies a pattern that we used to extract input parameters from the source code of the Moodle web application. The analyzer makes a best-effort attempt to find all occurrences of function invocations of `optional_param` having two parameters. The value in the first argument is recorded, and the second argument is a "don't care" that is ignored. The analyzer can capture the names of input parameters in a similar way when the input parameter is accessed via an array.

To perform the pattern matching itself, the analyzer transforms the pattern and the PHP script to be analyzed into an abstract syntax tree (AST). Then, the tries to match the pattern AST against the AST for the PHP script. For each match found in the source code, the analyzer then traverses the script's control flow graph (CFG) to check whether the match is reachable from the entry point of the script. For example, when an `optional_param` function invocation is observed, the analyzer checks whether a potential call chain exists from the invocation site to the script entry point. CFG traversal is recursive, including inclusions of other PHP files using the `require` and `include` statements.

Runtime enforcement: The runtime component is implemented as a PHP wrapper that is executed prior to invoking a PHP script using PHP's `autoprepend` mechanism. The PHP XMLReader library is used to parse input validation policies. The validation script checks the contents of all possible input vectors using the validation routines corresponding to each parameter's learned type.

E. Discussion

The IPAAS approach has the desirable property that, as opposed to automated output sanitization, it can be applied to virtually any language or development framework. IPAAS is can be deployed in an automated and transparent way such that the developer need not be aware that their application has been augmented with more rigorous input validation. While the potential for false positives does exist, our evaluation results in Section IV suggest that this would not be a major problem in practice.

However, our current implementation of IPAAS has a number of limitations. First, type learning can fail in the presence of custom query string formats. In this case, the

Application	PHP Files	Lines of Code
Joomla 1.5	450	128930
Moodle 1.6.1	1352	365357
Mybb 1.0	152	42989
PunBB 1.2.11	70	17374
Wordpress 1.5	125	29957

Table II
PHP APPLICATIONS USED IN OUR EXPERIMENTS.

IPAAS parameter extractor might not be able to reliably parse parameter key-value pairs.

Second, the prototype implementation of the static analyzer is fairly rudimentary. For instance, it cannot infer parameter names from variables or function invocations. Therefore, if an AST pattern is matched and the argument that is to be recorded is a non-terminal (e.g., a variable or function invocation), then the parameter name cannot be identified. In these cases, the location of the function invocation is stored along with a flag indicating that an input parameter was accessed in a dynamic way. This allows the developer the opportunity to identify the names of the input parameters manually after the analyzer has terminated, if desired.

IV. EVALUATION

To assess the effectiveness of our approach in preventing input validation vulnerabilities, we tested our IPAAS prototype on five real-world web applications shown in Table II. Each application is written in PHP, and the versions we tested contain many known, previously-reported XSS and SQL injection vulnerabilities.

To run our prototype, we created a development environment by importing each application as a project in Eclipse version 3.7 (Indigo) with PHP Development Tools (PDT) version 3.0 installed.

A. Vulnerabilities

Before starting our evaluation, we extracted the list of vulnerable parameters for reach application by analyzing the vulnerability reports stored in the Common Vulnerabilities and Exposures (CVE) database hosted by NIST [20]. For each extracted parameter, we manually verified the existence of the vulnerability in the corresponding application. In addition, we manually determined the data type of the vulnerable parameter.

Table III summarizes the results of the manual analysis, and shows, for each web application, the number of vulnerable parameters having a particular data type. The dataset resulting from this analysis contains 109 XSS and 120 SQL injection vulnerable parameters.

According to Table III, more than half of the SQL injections are associated with integer parameters, while the majority of the XSS vulnerabilities are exploited through the use of parameters of type `word`. Interestingly, only a relatively small number of vulnerabilities are caused by `free-text` or similarly unconstrained parameters. This supports our hypothesis that IPAAS can be used in practice to automatically prevent the majority of input validation vulnerabilities.

B. Automated Parameter Analysis

In order to automatically label parameters with types, IPAAS requires a training set containing examples of benign requests submitted to the web application. We collected this input data by manually exercising the web application and providing valid data for each parameter.

The results of our automated analysis are summarized in Table IV. For each application, the table reports the number of vulnerable parameters having a particular type. The results show that less than half of the parameters could be identified automatically. For most, our system was able to assign the correct type. However, in a few cases, the parameter was part of a request or serialized in a response, but had no value assigned to it. Hence, the type could not be identified. These parameters are reported as having type `unknown`.

Finally, IPAAS wrongly assigned the type `boolean` instead of `integer` to two XSS and four SQL injection vulnerable parameters. These misclassifications are caused by the overlap between `boolean` and `integer` validators. In fact, parameters having values of "0" and "1" can be considered of type `boolean` as well as `integer` (i.e., if only the values "0" and "1" are observed during training, the analysis engine gives priority to the type `boolean`). Collecting more data for each parameter by exercising the same functionality of a web application multiple times can result in different values for the same parameter. Hence, collecting more training data would increase the probability that our algorithm makes the correct classification.

C. Static Analyzer

To improve the detection ratio of the vulnerable parameters, we ran our static analyzer on the source code of each application.

Table V shows the number of vulnerable parameters that were identified with the help of the static analyzer. The tool was able to find 86% of the XSS and 87% of the SQL injection affected parameters. By comparing these input parameters with the ones that were detected by the analysis engine, we see that 26% of the XSS and 51% of the SQL injection affected parameters were missed by the analysis engine, but were found by the static analyzer. Hence, the static analyzer component can help in achieving a larger coverage of the type analysis, and, thus, help prevent a larger number of vulnerabilities.

Based on these results, we collected more input data by testing the functionality of each web application using the

Parameter Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	5	10	11	14	16	2	5	0	39 (36%)	30 (25%)
`integer`	1	7	0	28	6	23	6	3	4	2	17 (16%)	63 (53%)
`free-text`	3	2	4	0	5	1	4	0	13	0	29 (27%)	3 (3%)
`boolean`	1	0	0	1	1	4	5	0	0	0	7 (6%)	5 (4%)
`token`	1	2	0	0	3	8	1	2	0	1	5 (5%)	13 (11%)
`words`	2	1	0	1	0	0	2	0	1	0	5 (5%)	2 (2%)
`URL`	0	0	0	0	1	0	1	0	3	0	5 (5%)	0 (0%)
`list`	0	0	0	1	1	2	1	1	0	0	2 (2%)	4 (5%)
Total	10	16	9	41	28	52	36	8	26	3	109	120

Table III
MANUALLY IDENTIFIED DATA TYPES OF VULNERABLE PARAMETERS IN FIVE LARGE WEB APPLICATIONS.

Parameter Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	1	0	5	7	12	1	1	0	21 (19%)	12 (10%)
`integer`	1	6	0	2	2	8	5	1	3	0	11 (10%)	17 (14%)
`free-text`	3	2	1	0	4	0	2	0	10	0	20 (18%)	2 (2%)
`boolean`	1	0	0	0	1^1	3^3	4	0	1^1	1^1	7 (6%)	4 (3%)
`token`	1	2	0	0	1	3	1	2	0	1	3 (3%)	8 (6%)
`words`	2	0	0	0	0	0	1	0	0	0	3 (3%)	0 (0%)
`URL`	0	0	0	0	1	0	0	0	1	0	2 (2%)	0 (0%)
`list`	0	0	0	0	0	1	0	0	0	0	0 (0%)	1 (1%)
`unknown`	0	0	2	0	2	1	1	0	1	0	6 (6%)	1 (1%)
Correctly Identified	10	14	4	2	15	20	26	4	16	1	71 (65%)	41 (34%)
Wrongly Identified	-	-	-	-	1	3	-	-	1	1	2 (1.8%)	4 (3.3%)

(*) number reported as superscript indicate the parameters identified with an incorrect type.

Table IV
TYPING OF VULNERABLE PARAMETERS IN FIVE LARGE WEB APPLICATIONS BEFORE STATIC ANALYSIS.

Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
Detected by static analysis	3	9	6	40	28	46	24	8	23	1	94 (86%)	104 (87%)
Missed during type analysis	0	2	2	37	10	18	10	4	6	0	28 (26%)	61(51%)

Table V
RESULTS OF ANALYZING THE CODE.

data from the static analyzer. Then, we ran IPAAS again to determine the data types of the newly discovered parameters, and we manually verified whether the types were correctly identified. The results are shown in Table VI. In this case, we obtained better coverage, with 87% of XSS and 86% of SQL injection affected parameters being properly identified. In addition, none of the parameters were misclassified.

Although the static analyzer helps significantly in achieving a higher coverage, a few parameters were still missed during analysis. This problem could be improved by employing a more precise static analysis. Also, we believe that unit testing might serve as an additional source of test input data to help improve IPAAS' coverage.

D. *Impact*

To assess the extent to which IPAAS is effective in preventing input validation vulnerabilities in practice, we manually tested whether it was still possible to exploit the aforementioned vulnerabilities while IPAAS was enabled. During our tests, we explored different ways to perform the attacks, and to evade possible sanitization and validation routines as reported by XSS and SQL cheatsheets available on the Internet.

Table VII shows the number of XSS and SQL injection vulnerabilities that are prevented by IPAAS. We observe that most of the SQL injection vulnerabilities and a large fraction of XSS vulnerabilities became impossible to exploit with the input validation policies that were automatically extracted in

Type	Joomla		Moodle		MyBB		PunBB		Wordpress		Total	
	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli	xss	sqli
`word`	2	4	4	7	10	10	15	1	5	0	36 (33%)	22 (18%)
`integer`	1	7	0	25	6	21	5	3	4	2	16 (15%)	58 (48%)
`free-text`	3	2	3	0	4	1	2	0	10	0	22 (20%)	3 (3%)
`boolean`	1	0	0	1	1	3	4	0	0	0	6 (6%)	4 (3%)
`token`	1	2	0	0	3	8	1	2	0	1	5 (5%)	13 (11%)
`words`	2	1	0	1	0	0	2	0	1	0	5 (5%)	2 (2%)
`URL`	0	0	0	0	1	0	0	0	2	0	3 (3%)	0 (0%)
`list`	0	0	0	0	0	1	0	0	0	0	0 (0%)	1 (1%)
`unknown`	0	0	0	0	1	0	1	0	0	0	2 (2%)	0 (0%)
Total	10	16	7	34	26	44	30	6	22	3	95 (87%)	103 (86%)

Table VI
TYPING OF VULNERABLE PARAMETERS IN FIVE LARGE WEB APPLICATIONS AFTER STATIC ANALYSIS.

Application	Vulnerabilities		Prevented Vulnerabilities	
	xss	sql	xss	sqli
Joomla	10	16	7 (70%)	14 (88%)
Moodle	9	41	4 (44%)	34 (83%)
MyBB	28	52	21 (75%)	43 (83%)
PunBB	36	8	27 (75%)	6 (75%)
Wordpress	26	3	12 (46%)	3 (100%)
Total	109	120	71 (65%)	100 (83%)

Table VII
THE NUMBER OF PREVENTED VULNERABILITIES IN VARIOUS LARGE WEB APPLICATIONS.

our last experiment in place.

The results of this analysis are consistent with our observation that the majority of input validation vulnerabilities on the web can be prevented by labeling the parameter with a data type that properly constrains the range of legitimate values. If a parameter is assigned to an unknown or unrestricted type such as `free-text`, our system will still accept arbitrary input. In these cases, the vulnerability is not prevented by our system.

The difference in the number of prevented XSS and SQL injection vulnerabilities is mainly due to the relatively large number of `integer` parameters that are vulnerable to SQL injection, while many XSS vulnerabilities are due to injections in `free-text` parameters. We believe that the large number of parameters vulnerable to SQL injection that correspond to the type `integer` is caused by the phenomenon that web applications frequently use integers to identify records.

V. RELATED WORK

In this section, we place IPAAS in the context of related work on web application security.

Input validation: Much work has been done that aims to mitigate the impact of malicious input data without changing the application's source code. Scott and Sharp [32] proposed an application-level firewall to prevent malicious input from reaching the web server. Their approach required a specification of constraints on different inputs, and compiled those constraints into a policy validation program. In contrast, our approach automatically learns the specification of constraints.

Automating the task of generating test vectors for exercising input validation mechanisms is also a topic explored in the literature. Sania [16] is a system to be used in the development and debugging phases. It automatically generates SQL injection attacks based on the syntactic structure of queries found in the source code and tests a web application using the generated attacks. Saxena et al. proposed Kudzu [28], which combines symbolic execution with constraint solving techniques to generate test cases with the goal of finding client-side code injection vulnerabilities in JavaScript code. Halfond et al. [7] use symbolic execution to infer web application interfaces to improve test coverage of web applications. Several papers propose techniques based on symbolic execution and string constraint solving to automatically generate XSS and SQL injection attacks and input generation for systematic testing of applications implemented in C [4], [14], [13]. We consider these mechanisms to be complementary to our approach, in that they could be used to automatically generate malicious input for "free-text" fields, or to create legitimate input for other fields during the type learning phase.

Attack detection and prevention: Different techniques have been proposed to detect the occurrence of XSS and SQL injection attacks in HTTP traffic [3], [17], [24], [25]. Intrusion detection systems such as Snort [25], are configured with a number of 'signatures' that support the detection of web-based attacks. These systems match patterns that are associated with known exploits against HTTP traffic obtained while monitoring web applications. Unfortunately, it is very difficult to keep the set of signatures up-to-date as new signatures must be developed when new attacks or modifications to previously known attacks are discovered. Anomaly-based intrusion detection systems [3], [17], [24] establish models describing the normal behavior of the monitored system and rely on these models to identify anomalous activity that may be associated to intrusions. The main advantage of anomaly detection systems compared to signature-based intrusion detection is that they can identify unknown attacks. While anomaly-based detection systems have the potential to protect web applications effectively against XSS and SQL injection attacks, they suffer from a large number of false positives. In contrast to anomaly-based detection systems, our approach employs static analysis to achieve a larger coverage of protected parameters to the web application.

Preventative techniques for mitigating XSS and SQL injection vulnerabilities focus either on client-side mechanisms, or on server-side mechanisms. Client-side or browser-based mechanisms such as Noxes [15], Noncespaces [5], or DSI [19] make changes to the browser infrastructure aiming to prevent the execution of injected scripts. Each of these approaches requires that end-users upgrade their browsers or install additional software; unfortunately, many users do not regularly upgrade their systems [34].

Many techniques focus on the prevention of injection attacks using runtime monitoring. For example, Wassermann and Su [33] propose a system that checks at runtime the syntactic structure of a query for a tautology. AMNESIA [8] checks the syntactic structure of queries at runtime against a model that is obtained through static analysis. XSSDS [11] is a system that aims to detect XSS attacks by comparing HTTP requests and responses. While these systems focus on preventing injection attacks by checking the integrity of queries or documents, we focus on input validation. Recent work has focused on automatically discovering parameter injection [1] and parameter tampering vulnerabilities [22].

Among server-side approaches, leveraging language type systems has been proposed as an XSS and SQL defense mechanism by Robertson et al [23]. In this approach, XSS attacks are prevented by generating HTTP responses from statically-typed data structures that represent web documents. During document rendering, context-aware sanitization routines are automatically applied to untrusted values. The approach requires that the web application constructs HTML content using special algebraic data types.

Recent work has also focused on the correct use of sanitization routines to prevent XSS attacks. Scriptgard [29] can automatically detect and repair mismatches between sanitization routines and context. In addition, it ensures the correct ordering of sanitization routines. Samuel et al. [27] propose a type-qualifier based mechanism that can be used with existing templating languages to achieve context-sensitive auto-sanitization. Both approaches only focus on preventing XSS vulnerabilities. As we focus on automatically identifying parameter data types, our approach can help identify other vulnerabilities such as SQL injection or, in principle, HTTP Parameter Pollution.

Vulnerability analysis: Static analysis as a tool for finding security-critical bugs in software has also received a great deal of attention. WebSSARI [10] was one of the first efforts to apply classical information flow techniques to web application security vulnerabilities, where the goal of the analysis is to check whether a sanitization routine is applied before data reaches a sensitive sink. Several static analysis approaches have been proposed for various languages [12], [18]. Unfortunately, due to the inherently dynamic nature of scripting languages, static analysis tools are often imprecise [37]. The IPAAS approach incorporates a static analysis component as well as a dynamic component to learn parameter types. While our prototype static analyzer is simple and imprecise, our evaluation results are nevertheless encouraging.

Runtime approaches to automatically harden web applications have been proposed for PHP [21] and Java [6]. Although these approaches can work at a finer-grained level than static analysis tools, they incur runtime overhead. Both approaches aim to detect missing sanitization functionality while our focus is on the validation of untrusted user input.

The XSS cheatsheet [26] is a list of XSS vectors that can be used to bypass many sanitization routines. Balzarotti et al. [2] show that web applications do not always implement correct sanitization routines. The BEK project [9] proposes a system and languages for checking the correctness of sanitizers.

VI. Conclusion

Web applications are popular targets on the Internet, and well-known vulnerabilities such as XSS and SQL injection are, unfortunately, still prevalent. Current mitigation techniques for XSS and SQL injection vulnerabilities mainly focus on some aspect of automated output sanitization. In many cases, these techniques come with a large runtime overhead, lack precision, or require invasive modifications to the client or server infrastructure.

In this paper, we identify automated input validation as an effective alternative to output sanitization for preventing XSS and SQL injection vulnerabilities in legacy applications, or where developers choose to use insecure legacy languages and frameworks. We present the IPAAS approach,

which improves the secure development of web applications by transparently learning types for web application parameters during testing, and automatically applying robust validators for these parameters at runtime. The evaluation of our implementation for PHP demonstrates that IPAAS can automatically protect real-world applications against the majority of XSS and SQL injection vulnerabilities with a low false positive rate.

Acknowledgments

The research leading to these results was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) from the PoSecCo project (contract N 216917) and contract N 257007, by the FFG - Austrian Research Promotion Agency from the COMET K1-project and by National Science Foundation grant CNS-1116777. This work has also been supported by the French National Research Agency through the CESSA and VAMPIRE projects. We would also like to thank Secure Business Austria for their support.

References

[1] M. Balduzzi, C. Torrano Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS'11, 8th Annual Network and Distributed System Security Symposium, 6-9 February 2011, San Diego, California, USA*, 02 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Saner: composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[3] D. Bolzoni and S. Etalle. Boosting web intrusion detection systems by inferring positive signatures. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems*, OTM '08, pages 938–955, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, 2009.

[6] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[7] W. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, Illinois, USA, July 2009.

[8] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, CA, USA, Nov 2005.

[9] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[11] M. Johns, B. Engelmann, and J. Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 335–344, Washington, DC, USA, 2008. IEEE Computer Society.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Oakland, CA, USA, 2006. IEEE Computer Society.

[13] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, pages 105–116, Chicago, IL, USA, July 21–23, 2009.

[14] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.

[15] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 330–337, Dijon, FR, 2006. ACM.

[16] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama. Sania: Syntactic and semantic analysis for automated testing against sql injection. In *ACSAC*, pages 107–117. IEEE Computer Society, 2007.

[17] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, Washington, DC, October 2003. ACM Press.

[18] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, Aug 2005.

[19] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, 2009.

[20] National Institute of Standards and Technology. National Vulnerability Database Version 2.2. http://nvd.nist.gov/, 2010.

[21] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[22] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V.N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *CCS'10: Proceedings of the 17th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2010.

[23] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th USENIX Security Symposium*, pages 283–298. USENIX Association, 2009.

[24] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[25] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[26] RSnake. Xss (cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html, 2009.

[27] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 587–600, New York, NY, USA, 2011. ACM.

[28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[29] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2011.

[30] T. Scholte, D. Balzarotti, and E. Kirda. Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, Bay Gardens Beach Resort, Saint Lucia, February 2011.

[31] T. Scholte, D. Balzarotti, W. Robertson, and E. Kirda. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *Proceedings of the 27th ACM Symposium On Applied Computing (SAC 2012)*, Riva del Garda, Italy., March 2012.

[32] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[33] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[34] W3Counter. Web browser market share trends. http://www.w3counter.com/trends, 2011.

[35] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, Jun 2007. ACM.

[36] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical report, UC Berkeley, 2011.

[37] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, 2006. USENIX Association.