Secure Code Generation for Web Applications

Martin Johns[1,2], Christian Beyerlein[3], Rosemaria Giesecke[1], and Joachim Posegga[2]

[1] SAP Research – CEC Karlsruhe
{martin.johns,rosemaria.giesecke}@sap.com
[2] University of Passau, Faculty for Informatics and Mathematics, ISL
{mj,jp}@sec.uni-passau.de
[3] University of Hamburg, Department of Informatics, SVS
9beyerle@informatik.uni-hamburg.de

Abstract. A large percentage of recent security problems, such as Cross-site Scripting or SQL injection, is caused by string-based code injection vulnerabilities. These vulnerabilities exist because of implicit code creation through string serialization. Based on an analysis of the vulnerability class' underlying mechanisms, we propose a general approach to outfit modern programming languages with mandatory means for explicit and secure code generation which provide strict separation between data and code. Using an exemplified implementation for the languages Java and HTML/JavaScript respectively, we show how our approach can be realized and enforced.

1 Introduction

The vast majority of today's security issues occur because of string-based code injection vulnerabilities, such as Cross-site Scripting (XSS) or SQL Injection.An analysis of the affected systems results in the following observation: All programs that are susceptible to such vulnerabilities share a common characteristics – They utilize the string type to assemble the computer code that is sent to application-external interpreters. In this paper, we analyse this vulnerability type's underlying mechanisms and propose a simple, yet effective extension for modern programming languages that reliably prevents the introduction of string-based code injection vulnerabilities.

1.1 The Root of String-Based Injection Vulnerabilities

Networked applications and especially web applications employ a varying amount of heterogeneous computer languages (see Listing 1), such as programming (e.g., Java, PHP, C#), query (e.g., SQL or XPATH), or mark-up languages (e.g., XML or HTML).

```
1 // embedded HTML syntax
2 out.write("<a href='http://www.foo.org'>go</a>");
3 // embedded SQL syntax
4 sql = "SELECT * FROM users";
```

Listing 1. Examples of embedded syntax

F. Massacci, D. Wallach, and N. Zannone (Eds.): ESSoS 2010, LNCS 5965, pp. 96–113, 2010.

This observation leads us to the following definition:

Definition 1 (Hosting/Embedded). *For the remainder of this paper we will use the following naming convention:*

- **Hosting language:** *The language that was used to program the actual application (e.g., Java).*
- **Embedded language:** *All other computer languages that are employed by the application (e.g., HTML, JavaScript, SQL, XML).*

Embedded language syntax is assembled at run-time by the hosting application and then passed on to external entities, such as databases or web browsers. String-based code injection vulnerabilities arise due to a mismatch in the programmer's intent while assembling embedded language syntax and the actual interpretation of this syntax by the external parser. For example, take a dynamically constructed SQL-statement (see Fig. 1.a). The application's programmer probably considered the constant part of the string-assembly to be the *code*-portion while the concatenated variable was supposed to add dynamically *data*-information to the query. The database's parser has no knowledge of the programmer's intent. It simply parses the provided string according to the embedded language's grammar (see Fig. 1.b). Thus, an attacker can exploit this discord in the respective views of the assembled syntax by providing data-information that is interpreted by the parser to consist partly of *code* (see Fig. 1.c).

In general all string values that are provided by an application's user on runtime should be treated purely as *data* and never be executed. But in most cases the hosting language does not provide a mechanism to explicitly generate embedded syntax. For this reason all embedded syntax is generated implicitly by string-concatenation and -serialization. Thus, the hosting language has no means to differentiate between user-provided dynamic *data* and programmer-provided embedded *code* (see Fig. 1.d). Therefore, it is the programmer's duty to make sure that all dynamically added *data* will not be parsed as *code* by the external interpreter. Consequently, if a flaw in the application's logic allows an inclusion of arbitrary values into a string segment that is passed as embedded syntax to an external entity, an attacker can succeed in injecting malicious *code*.

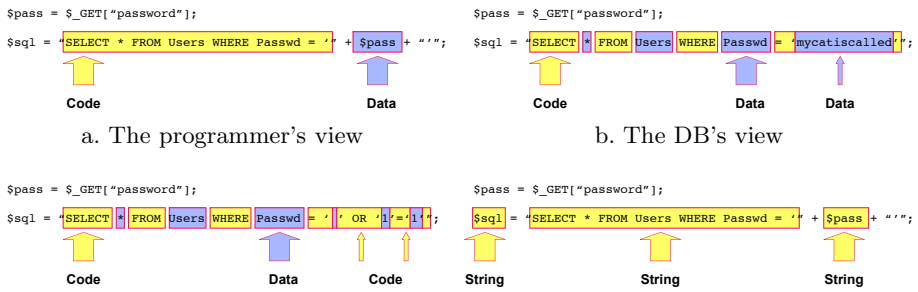


Fig. 1. Mismatching views on code

This bug pattern results in many vulnerability types, such as XSS, SQL injection, directory traversal, shell command injection, XPath injection or LDAP injection.

A considerable amount of work went into addressing this problem by approaches which track the flow of untrusted data through the application (see Sec. 4). However, with the exception of special domain solutions, the causing practice of using strings for code assembly has not been challenged yet.

1.2 The String Is the Wrong Tool!

As motivated above, assembly of computer code using the string datatype is fundamentally flawed. The string is a "dumb" container type for sequences of characters. This forces the programmer to create embedded syntax *implicitly*. Instead of clearly stating that he aims to create a `select`-statement which accesses a specific table, he has to create a stream of characters which hopefully will be interpreted by the database in the desired fashion. Besides being notoriously error-prone, this practice does not take full advantage of the programmer's knowledge. He knows that he wants to create a `select`-statement with very specific characteristics. But the current means do not enable him to state this intent *explicitly* in his source code.

1.3 Contributions and Paper Outline

In this paper we propose to add dedicated capabilities to the hosting language that

- allow secure creation of embedded syntax through
- enforcing explicit creation of code semantics by the developer and
- providing strict separation between embedded data and code.

This way, our method reliably prevents string-based code injection vulnerabilities. Furthermore, our approach is

- applicable for any given embedded language (e.g., HTML, JavaScript, SQL),
- can be implemented without significant changes in the hosting language,
- and mimics closely the current string-based practices in order to foster acceptance by the developer community.

The remainder of the paper is organised as follows. In Section 2 we describe our general objectives, identify our approach's key components, and discuss how these components can be realised. Then, in Section 3 we document our practical implementation and evaluation. For this purpose, we realised our method for the languages Java and HTML/JavaScript. We end the paper with a discussion of related work (Sec. 4) and a conclusion (Sec. 5).

2 Concept Overview

In this section, we propose our language-based concept for assembly of embedded syntax which provides robust security guarantees.

2.1 Keeping the Developers Happy

Before describing our approach, in this section we list objectives which we consider essential for any novel language-based methodology to obtain acceptance in the developer community.

Objectives concerning the hosting language: Foremost, the proposed concepts should not depend on the characteristics of a specific hosting language. They rather should be applicable for any programming language in the class of procedural and object-orientated languages[1]. Furthermore, the realisation of the concepts should also not profoundly change the hosting language. Only aspects of the hosting language that directly deal with the assembly of embedded code should be affected.

Objectives concerning the creation of embedded syntax: The specific design of every computer language is based on a set of paradigms that were chosen by the language's creators. These paradigms were selected because they fitted the creator's design goals in respect to the language's scope. This holds especially true for languages like SQL that were not designed to be a general purpose programming language but instead to solve one specific problem domain. Therefore, a mechanism for assembling such embedded syntax within a hosting language should aim to mimic the embedded language as closely as possible. If the language integration requires profound changes in the embedded syntax it is highly likely that some of the language's original design paradigms are violated. Furthermore, such changes would also cause considerable training effort even for developers that are familiar with the original embedded language.

Finally, string operations, such as string-concatenation, string-splitting, or search-and-replace, have been proven in practice to be a powerful tool for syntax assembly. Hence, the capabilities and flexibility of the string type should be closely mimicked by the newly introduced methods.

2.2 Key Components

We propose a reliable methodology to assemble embedded language syntax in which all code semantics are created explicitly and which provides a strict separation between data and code. In order to do so, we have to introduce three key components to the hosting language and its run-time environment: A dedicated datatype for embedded code assembly, a suitable method to integrate the embedded language's syntax into the hosting code, and an external interface which communicates the embedded code to the external entity (see Fig. 2).

Datatype: We have to introduce a novel datatype to the hosting language that is suitable to assemble/represent embedded syntax and that guarantees strict separation between data and code according to the developer's intent. In the context of this document we refer to such a datatype as **ELET**, which is short for *Embedded Language Encapsulation Type*.

[1] To which degree this objective is satisfiable for functional and logical programming languages has to be determined in the future.

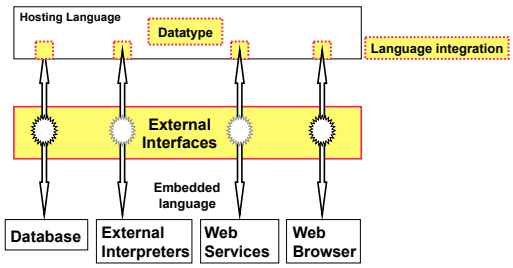


Fig. 2. Key components of the proposed approach

Language integration: To fill the ELET datatype, suitable means have to be added to the hosting language which enable the developer to pass embedded syntax to the datatype. As motivated above these means should not introduce significant changes to the embedded language's syntax.

External interface: For the time being, the respective external entities do not yet expect the embedded code to be passed on in ELET form. For this reason, the final communication step is still done in character based form. Therefore, we have to provide an external interface which securely translates embedded syntax which is encapsulated in an ELET into a character-based representation. Furthermore, all legacy, character-based interfaces to the external entities have to be disabled to ensure our approach's security guarantees. Otherwise, a careless programmer would still be able to introduce code injection vulnerabilities.

In the following sections we show how these three key components can be realised.

2.3 The Embedded Language Encapsulation Type (ELET)

This section discusses our design decisions concerning the introduced datatype and explains our security claims.

As shown in Sec. 1.1, the root cause for string-based code injection vulnerabilities is that the developer-intended data/code-semantics of the embedded syntax are not correctly enforced by the actual hosting code which is responsible for the syntax assembly. Therefore, to construct a methodology which reliably captures the developer's intent and provides strict separation between data and code, it is necessary to clarify two aspects of embedded syntax assembly:

For one, an intended classification of syntax into *data* and *code* portions implies that an underlying segmentation of individual syntactical language elements exist. Hence, the details of this segmentation have to be specified. Furthermore, after this specification step, a rationale has to be built which governs the assignment of the identified language elements into the *data*/*code*-classes.

ELET-internal representation of the embedded syntax: The main design goal of the ELET type is to support a syntax assembly method which bridges the current gap between string-based code creation and the corresponding parsing process of the application-external parsers. For this reason, we examined the

mechanisms of parsing of source code: In general, the parsing is, at least, a two step process. First a lexical analysis of the input character stream generates a stream of tokens. Then, a syntactical analysis step matches these tokens according to the language's formal grammar and aligns them into tree structures such as parse-trees or abstract syntax-trees. The leafs of such trees are the previously identified tokens, now labeled according to their grammar-defined type (e.g., JavaScript defines the following token types: keyword-token, identifier-token, punctuator-token, and data-value).

Hence, we consider regarding an embedded language statement as a stream of labeled tokens to be a suitable level of abstraction which allows us to precisely record the particularities of the embedded syntax. Consequently, the ELET is a container which holds a flat stream of labeled language tokens. See Listing 2 for an exemplified representation of Sec. 1.1's SQL.

```
1 $sql = {select-token, meta-char(*), from-token, tablename-token(Users),
2         where-token, fieldname-token(Passwd), metachar(=), metachar('),
3         stringliteral(mycatiscalled), metachar(')}
```

Listing 2. ELET-internal representation of Fig. 1.b's SQL statement

Identification of language elements: An analysis of common embedded languages, such as HTML, SQL, or JavaScript, along with an investigation of the current practice of code assembly yields the following observation: The set of existing token-types can be partitioned into three classes - static tokens, identifier tokens, and data literals (see Fig. 3):

Static tokens are statically defined terminals of the embedded language's grammar. Static tokens are either language keywords (such as SQL instructions, HTML tag- and attribute-names, or reserved JavaScript keywords) or metacharacters which carry syntactical significance (such as `*` in SQL, `<` in HTML, or `;` in JavaScript).

Identifier tokens have values that are not statically defined within the language's grammar. However, they represent semi-static elements of the embedded code, such as names of SQL tables or JavaScript functions. As already observed and discussed, e.g., in [2] or [6], such identifiers are statically defined at compile-time of the hosting code. I.e., the names of used JavaScript functions are not dynamically created at run-time – they are hard-coded through constant string-literals in the hosting application.

Data literals represent values which are used by the embedded code. E.g., in Fig. 1.b the term "mycatiscalled" is such a literal. As it can be obtained from Fig. 1.a the exact value of this literal can change every time the hosting code is executed. Hence, unlike static and identifier tokens the value of data literals are dynamic at run-time.

Rules for secure assembly of embedded syntax: As previously stated, our methodology relies on *explicit* code creation. This means, for instance, the only way for a developer to add a static token representing a `select`-keyword to a

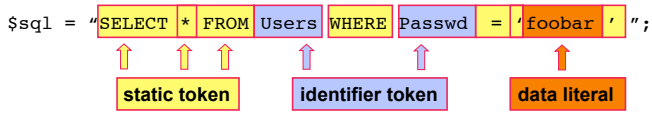


Fig. 3. Examples of the identified token classes

SQL-ELET is by explicitly stating his intent in the hosting source code. Consequently, the ELET has to expose a set of disjunct interfaces for adding members of each of the three token-classes, e.g. implemented via corresponding APIs. For the remainder of this paper, we assume that these interfaces are in fact implemented via API-calls. Please note: This is not a defining characteristic of our approach but merely an implementation variant which enables easy integration of the ELET into existing languages.

We can sufficiently model the three token-classes by designing the ELET's API according to characteristics identified above regarding the define-time of the tokens' values (see also Listing 3):

- API calls to add a static token to the ELET container are themselves completely static. This means, they do not take any arguments. Hence, for every element of this token-class, a separate API call has to exist. For instance, to add a `select`-keyword-token to a SQL-ELET a corresponding, dedicated `addSelectToken()`-method would have to be invoked.
- Identifier tokens have values which are not predefined in the embedded language's grammar. Hence, the corresponding ELET interface has to take a parameter that contains the token's value. However, to mirror fact that the values of such tokens are statically defined at compile-time and do not change during execution, it has to be enforced that the value is a constant literal. Furthermore, the embedded language's grammar defines certain syntactic restrictions for the token's value, e.g., the names of JavaScript variables can only be composed using a limited set of allowed characters excluding whitespace. As the arguments for the corresponding API methods only accept constant values these restrictions can be checked and enforced at compile-time.
- The values of data literals can be defined completely dynamically at runtime. Hence, the corresponding API methods do not enforce compile-time restrictions on the values of their arguments (which in general consist of string or numeric values).

```
1 sqlELET.addSelectToken();             // no argument
2 [...]
3 sqlELET.addIdentifierToken("Users"); // constant argument
4 ...
5 sqlELET.addStringLiteral(password);  // dynamic argument
6 [...]
```

Listing 3. Exemplified API-based assembly of Fig. 1.b's SQL statement

Through this approach, we achieve reliable protection against string-based code injection vulnerabilities: All untrusted, attacker-controlled values enter the

application as either string or numeric values, e.g., through HTTP parameters or request headers. In order to execute a successful code injection attack the adversary has to cause the hosting application to insert parts of his malicious values into the ELET in a syntactical *code* context, i.e., as static or identifier tokens. However, the ELET interfaces which add these token types to the container do not accept dynamic arguments. Only *data* literals can be added this way. Consequently, a developer is simply not able to write source code which involuntarily and implicitly creates embedded *code* from attacker-provided string values. Hence, string-based code injection vulnerabilities are impossible as the ELET maintains at all times strict separation between the embedded *code* (the static and identifier tokens) and *data* (the data literal tokens).

2.4 Language Integration

While the outlined API-based approach fully satisfies our security requirements, it is a clear violation of our developer-oriented objective to preserve the embedded language's syntactic nature (see Sec. 2.1). A direct usage of the ELET API in a non-trivial application would result in source code which is very hard to read and maintain. Consequently, more suitable methods to add embedded syntax to the ELET have to be investigated.

A direct integration of the embedded syntax into the hosting language, for instance through a combined grammar, would be the ideal solution (see Listing 4). In such a case, the developer can directly write the embedded syntax into the hosting application's source code. First steps in this direction have been taken by Meijer et al. with Xen [14] and LINQ [13] which promote a subset of SQL and XML to be first class members of C# However, this approach requires significant, non-trivial changes in the hosting language which are not always feasible. Also achieving full coverage of all features of the embedded language this way is still an open problem. Even advanced projects such as LINQ only cover a subset of the targeted languages and omit certain, sophisticated aspects which are hard to model within C#'s functionality.

```
1 sqlELET s = SELECT * FROM Users;
```

Listing 4. Direct integration of the embedded syntax

Instead, we propose a light-weight alternative which uses a source-to-source preprocessor: The embedded syntax is directly integrated into the hosting application's source code, unambiguously identified by predefined markers, e.g. $$ (see Listing 5). Before the application's source code is compiled (or interpreted), the preprocessor scans the source code for the syntactical markers. All embedded syntax that can be found this way is parsed accordingly to the embedded language's grammar and for each of the identified language tokens a corresponding API-call is added to the hosting source code (see Listing 6). The resulting API-based representation is never seen by the developer. He solely has to use unmodified embedded syntax. To add data values through hosting variables to the embedded syntax and offer capabilities for more dynamic code assembly (e.g.,

selective concatenation of syntax fragments) a simple meta-syntax is provided (see Sec. 3.2 for details).

```
1 sqlELET s = $$ SELECT * FROM Users $$
```

Listing 5. Preprocessor-targeted integration of embedded syntax

```
1 sqlELET s.addSelectToken().addStarMetachar().addFromToken()
2          .addIdentifierToken("Users");
```

Listing 6. Resulting API representation generated by the preprocessor

Compared to direct language integration, this approach has several advantages. For one, besides the adding of the ELET type, no additional changes to the original hosting language have to be introduced. The preprocessor is completely separate from the hosting language's parsing process. This allows flexible introduction of arbitrary embedded languages. Furthermore, achieving complete coverage of all features of the embedded language is not an issue. The only requirements are that the preprocessor covers the complete parsing process of the embedded language and the ELET provides interfaces for all existing tokens. Hence, covering obscure or sophisticated aspects of the embedded language is as easy as providing only a subset of the language.

However, our approach has also a drawback: The source code that is seen by the hosting application's parser differs from the code that the developer wrote. If parsing errors occur in source code regions which were altered by the preprocessor, the content of the parser's error message might refer to code that is unknown to the programmer. For this reason, a wrapped parsing process which examines the error messages for such occurrences is advisable.

2.5 External Interface

Finally, before communicating with the external entity (e.g., the database), the ELET-contained embedded syntax has to be serialized into a character-based representation. This is done within the external interface. From this point on, all meta-information regarding the tokens is lost. Therefore, the reassembly into a character-based representation has to be done with care to prevent the reintroduction of code injection issues at the last moment. The embedded syntax is provided within the ELET in a semi-parsed state. Hence, the external interface has exact and reliable information on the individual tokens and their *data*/*code*-semantics. Enforced by the ELET's restrictive interfaces, the only elements that may be controlled by the adversary are the *data* values represented by data literal tokens. As the exact context of these values within the embedded syntax is known to the external entity, it can reliably choose the correct encoding method to ensure the *data*-status of the values (see Sec. 3.3 for a concrete implementation).

2.6 Limitations

Before we document our practical implementation of our approach, we discuss the limitations of our methodology:

Run-time code evaluation: Certain embedded languages, such as JavaScript, provide means to create code from string values at run-time using function like `eval()`. Careless usage of such functionality can lead to string-based code injection vulnerabilities which are completely caused by the embedded code, such as DOM-based XSS [9]. The protection capabilities of our approach are limited to code assembly within the hosting language. To provide protection against this specific class of injection vulnerabilities, an ELET type has to be added to the embedded language and the `eval()`-functionality has to be adapted to expect ELET-arguments instead of string values.

Dynamically created identifier tokens: As explained in Sec. 2.3 the values of identifier tokens have to be defined through constant values. However, sometimes in real-life code one encounters identifiers which clearly have been created at run-time. One example are names of JavaScript variables which contain an enumeration component, such as `handler14`, `handler15`, etc. As this is not allowed by our approach, the developer has to choose an alternative solution like his purpose, e.g. arrays.

Permitting dynamic identifiers can introduce insecurities as it may lead to situations in which the adversary fully controls the value of an identifier token. Depending on the exact situation this might enable him to alter the control or data flow of the application, e.g. by exchanging function or variable names.

3 Practical Implementation and Evaluation

3.1 Choosing an Implementation Target

To verify the feasibility of our concepts, we designed and implemented our approach targeting the embedded languages HTML/JavaScript[2] and the hosting language Java. We chose this specific implementation target for various reasons: Foremost, XSS problems can be found very frequently, rendering this vulnerability-class to be one of the most pressing issues nowadays.Furthermore, as HTML and JavaScript are two independent languages with distinct syntaxes which are tightly mixed within the embedded code, designing a suiting preprocessor and ELET-type is interesting. Finally, reliably creating secure HTML-code is not trivial due to the lax rendering process employed by modern web browsers.

3.2 API and Preprocessor Design

The design of the ELET API for adding the embedded syntax as outlined in Sec. 2.3 was straightforward after assigning the embedded languages' elements

[2] NB: The handling of Cascading Style Sheets (CSS), which embody in fact a third embedded language, is left out in this paper for brevity reasons.

to the three distinct token types. Also, implementing the language preprocessor to translate the embedded code into API calls was in large parts uneventful. The main challenge was to deal with situations in which the syntactical context changed from HTML to JavaScript and vice versa. Our preprocessor contains two separate parsing units, one for each language. Whenever a HTML element is encountered that signals a change of the applicable language, the appropriate parser is chosen. Such HTML elements are either opening/closing `script`-tags or HTML attributes that carry JavaScript-code, e.g., event handlers like `onclick`.

As motivated in Sec. 2.1, we aim to mimic the string type's characteristics as closely as possible. For this purpose, we introduced a simple preprocessor meta-syntax which allows the programmer to add hosting values, such as Java strings, to the embedded syntax and to combine/extend ELET instances (see Listings 7 and 8). Furthermore, we implemented an iterator API which allows, e.g., to search a ELET's data literals or to split an ELET instance.

```
1 HTMLElet h $=$ Hello <b> $data(name)$ </b>, nice to see you! $$
```

Listing 7. Combination of embedded HTML with the Java variable `name`

3.3 Adding an External Interface for HTML Creation to J2EE

In order to assemble the final HTML output the external interface iterates through the ELET's elements and passes them to the output buffer. To deterministically avoid XSS vulnerabilities, all encountered data literals are encoded into a form which allows the browser to display their values correctly without accidentally treating them as *code*. Depending on the actual syntactical context a given element appears in an HTML page, a different encoding technique has to be employed:

- HTML: All meta-characters, such as `<`, `>`, `=`, `"` or `'`, that appear in data literals within an HTML context are encoded using HTML encoding ("`&...;`") to prevent the injection of rogue HTML tags or attributes.
- JavaScript-code: JavaScript provides the function `String.fromCharCode()` which translates a numerical representation into a corresponding character. To prevent code injection attacks through malicious string-values, all data literals within a syntactical JavaScript context are transformed into a representation consisting of concatenated `String.fromCharCode()`-calls.
- URLs: All URLs that appear in corresponding HTML attribute values are encoded using URL encoding ("`%..`").

In our J2EE based implementation the external interface's tasks are integrated into the application server's request/response handling. This is realized by employing J2EE's filter mechanism to wrap the `ServletResponse`-object. Through this wrapper-object servlets can obtain an `ELETPrinter`. This object in turn provides an interface which accepts instantiated ELETs as input (see Fig. 4.a). The serialized HTML-code is then passed to the original `ServletResponse`-object's

output buffer. Only input received through the `ELETPrinter` is included in the final HTML-output. Any values that are passed to the output-stream through legacy character-based methods is logged and discarded. This way we ensure that only explicitly generated embedded syntax is sent to the users web browsers.

Implementing the abstraction layer in the form of a J2EE filter has several advantages. Foremost, no changes to the actual application-server have to be applied - all necessary components are part of a deployable application. Furthermore, to integrate our abstraction layer into an existing application only minor changes to the application's `web.xml` meta-file have to be applied (besides the source code changes that are discussed in Section 3.4).

3.4 Practical Evaluation

We successfully implemented a preprocessor, ELET-library, and abstraction layer for the J2EE application server framework. To verify our implementation's protection capabilities, we ran a list of well known XSS attacks [3] against a specifically crafted test application. For this purpose, we created a test-servlet that blindly echos user-provided data back into various HTML/JavaScript data- and code-contexts (see Fig. 8 for an excerpt).

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException
{
     String bad = req.getParameter("bad");
    [...]
     HTMLElet h $=$ <h3>Protection test</h3> $$
     h $+$ Text: $data(bad)$ <br /> $$
     h $+$ Link: <a href="$data(bad)$">link</a> <br /> $$
     h $+$ Script: <script>document.write($data(bad)$);</script><br /> $$
    [...]
     EletPrinter.write(resp, h);  // Writing the ELET to the output buffer
     resp.getWriter().println(bad); // Is the legacy interface disabled?
}
```

Listing 8. Test-servlet for protection evaluation (excerpt)

Furthermore, to gain experiences on our approach's applicability in respect to non-trivial software we modified an existing software project to use our methodology. Porting an application to our approach requires to locate every portion of the application's source code which utilize strings to create embedded code. Such occurrences have to be changed to employ ELET semantics instead. Therefore, depending on the specific application, porting an existing code-base might prove to be difficult. We chose JSPWiki [5] as a porting target. JSPWiki is a mature J2EE based WikiWiki clone which was initially written by Janne Jalkanen and is released under the LGPL. More precisely, we chose version 2.4.103 of the software, a release which suffers from various disclosed XSS vulnerabilities [10]. The targeted version of the code consists of 365 java/jsp files which in total contain 69.712 lines of source code.

The porting process was surprisingly straightforward. JSPWiki's user-interface follows in most parts a rather clean version of the Model-View-Controller (MVC)

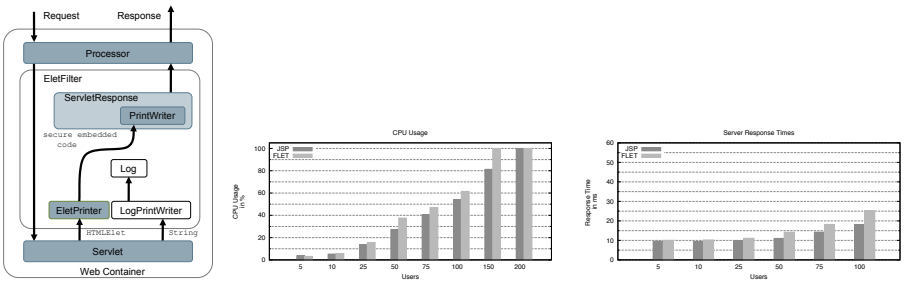
pattern which aided the porting process. Besides the user-interface also the application's Wiki-markup parser and HTML-generator had to be adapted. It took a single programmer about a week to port the application's core functionality. In total 103 source files had to be modified.

As expected, all documented XSS vulnerabilities [10] did not occur in the resulting software. This resolving of the vulnerabilities was solely achieved by the porting process without specifically addressing the issues in particular.

Finally, to gain first insight of our approach's runtime behaviour, we examined the performance of the prototype. For this purpose, we benchmarked the unaltered JSP code against the adapted version utilizing the ELET paradigm. The performance tests were done using the HP LoadRunner tool, simulating an increasing number of concurrent users per test-run. The benchmarked applications were served by an Apache Tomcat 5.5.20.0 on a 2,8 GHz Pentium 4 computer running Windows XP Professional. In situations with medium to high server-load, we observed an overhead of maximal 25% in the application's response times (see Fig. 4.c). Considering that neither the ELET's nor the external interface's implementation have been specifically optimized in respect to performance, this first result is very promising. Besides streamlining the actual ELET implementation, further conceptual options to enhance the implementation's performance exist. For instance, integrating the abstraction layer directly into the application server, instead of introducing it via object wrappers would aid the overall performance.

4 Related Work

In the last decade extensive research concerning the prevention of code injection attacks has been conducted. In this section we present selected related approaches. In comparison to our proposed solution, most of the discussed techniques have the limitations that they are not centrally enforceable and/or are prone to false positives/negatives. Both characteristics do not apply to our technique.



a. Concept b. CPU overhead c. Response times

Fig. 4. Implementation of the J2EE EletFilter

Manual protection and special domain solutions: The currently used strategy against XSS attacks is manually coded input filtering and output encoding. As long as unwanted HTML and JavaScript code is properly detected and stripped from all generated web pages, XSS attacks are impossible. However, implementing these techniques is a non-trivial and error prone task which cannot be enforced centrally, resulting in large quantities of XSS issues. In order to aid developers to identify XSS and related issues in their code, several information-flow based approaches for static source code analysis have been discussed, e.g., [4], [11], [7], or [23]. However, due to the undecidable nature of this class of problems such approaches suffer from false positives and/or false negatives.

Manual protection against SQL injection suffers from similar problems as observed with XSS. However, most SQL interpreters offer *prepared statements* which provide a secure method to outfit static SQL statements with dynamic data. While being a powerful migration tool to avoid SQL injection vulnerabilities, prepared statements are not completely bulletproof. As dynamic assembly of prepared statements is done using the string type, injection attacks are still possible at the time of the initial creation of the statement [22]. Furthermore, methods similar to prepared statements for most other embedded languages besides SQL do not exist yet. Therefore, dynamic assembly of embedded code in these languages has to employ similar mitigating strategies as mentioned in the context of XSS, with comparable results.

Type based protection: In concurrent and independent work [19], Robertson and Vigna propose a type based approach which is closely related to our ELET paradigm. They introduce dedicated datatypes that model language features of HTML and SQL. This way they enforce separation of the *content* and *structure* of the created syntax, two concepts that are similar to this paper's terms *data* and *code*.

However, their approach is not easily extensible to general purpose programming languages, such as JavaScript. Consequently, cross-site scripting caused by insecure dynamic assembly of JavaScript code (e.g., see Listing 8, line 9) is not prevented. In comparison, our approach is language independent and, thus, covers dynamically assembled JavaScript. Furthermore, notably absent from the paper is a discussion concerning the actual construction of the embedded code. For example, it is not explained, how the programmer creates HTML syntax in practice. As previously discussed, we consider such aspect to be crucial in regard to acceptance of the developer community.

Taint propagation: Run-time taint propagation is regarded to be a powerful tool for detecting string-based code injection vulnerabilities. Taint propagation tracks the flow of untrusted data through the application. All user-provided data is "tainted" until its state is explicitly set to be "untainted". This allows the detection if untrusted data is used in a security sensible context. Taint propagation was first introduced by Perl's taint mode. More recent works describe finer grained approaches towards dynamic taint propagation. These techniques

allow the tracking of untrusted input on the basis of single characters. In independent concurrent works [16] and [18] proposed fine grained taint propagation to counter various classes of injection attacks. [2] describes a related approach ("positive tainting") which, unlike other proposals, is based on the tracking of trusted data. Furthermore, based on dynamic taint propagation, [21] describe an approach that utilizes specifically crafted grammars to deterministically identify code injection attempts. Finally, [24] proposes a fine grained taint mechanism that is implemented using a C-to-C source code translation technique. Their method detects a wide range of injection attacks in C programs and in languages which use interpreters that were written in C. To protect an interpreted application against injection attacks the application has to be executed by a recompiled interpreter.

Compared to our approach, dynamic taint propagation provides inferior protection capabilities. Taint-tracking aims to prevent the *exploitation* of injection vulnerabilities while their fundamental causes, string-based code assembly and the actual vulnerable code, remain unchanged. Therefore, in general the sanitazion of the tainted data still relies on string operations. The application has to "untaint" data after applying manually written validation and encoding function, a process which in practice has been proven to be non-trivial and error-prone. This holds especially true in situations where limited user-provided HTML is permitted. E.g., no taint-tracking solution would have prevented the `myspace.com` XSS that was exploited by the Samy-worm [8]. In our approach, even in cases where user-provided HTML is allowed, such markup has to be parsed from the user's data and recreated explicitly using ELET semantics, thus, effectively preventing the inclusion of any unwanted code. Furthermore, unlike our approach taint-tracking is susceptible to second-order code injection vulnerabilities [17] due to its necessary classification of data origins as either *trusted* or *untrusted*. In the case of second-order code injection the attacker is able to reroute his attack through a trusted component (e.g., temporary storage of an XSS attack in the DB).

Embedded language integration: Extensive work has been done in the domain of specifically integrating a certain given embedded language into hosting code. Especially SQL and XML-based languages have received a lot of attention. However, unlike our approach, the majority of these special purpose integration efforts neither can be extended to arbitrary embedded languages, nor have been designed to prevent code injection vulnerabilities. In the remainder of this section we describe selected publications that share similarities with our approach:

As previously discussed in Section 2.3, LINQ [13] promotes subsets of XML and SQL to be first class members of C#. Furthermore, E4X [20] is a related approach which integrates XML into JavaScript. The applied techniques are entirely data-centric with the goal to soften the object-relational impedance mismatch [14]. Due to this characteristic, their approach is not easily extensible to procedural embedded languages, such as JavaScript. In comparison, our approach is completely independent from the characteristics of the embedded

language. [12] describes SQL DOM. A given database schema can be used to automatically generate a SQL Domain Object Model. This model is transformed to an API which encapsulates the capabilities of SQL in respect to the given schema, thus eliminating the need to generate SQL statements with the string datatype. As every schema bears a schema-specific domain object model and consequently a schema-specific API, every change in the schema requires a regeneration of the API. Finally, SQLJ [1] and Embedded SQL [15], two independently developed mechanisms to combine SQL statements either with Java or C respectively, employ a simple preprocessor. However, unlike our proposed approach these techniques only allow the inclusion of static SQL statements in the source code. The preprocessor then creates hosting code that immediately communicates the SQL code to the database. Thus, dynamic assembly and processing of embedded syntax, as it is provided in our proposed approach via the ELET, is not possible.

5 Conclusion

In this paper we proposed techniques to enhance programming languages with capabilities for secure and explicit creation of embedded code. The centerpiece of our syntax-assembly architecture is the ELET (see Section 2.3), an abstract datatype that allows the assembly and processing of embedded syntax while strictly enforcing the separation between *data* and *code* elements. This way injection vulnerabilities that are introduced by implicit, string-serialization based code-generation become impossible. To examine the feasibility of the proposed approach, we implemented an integration of the embedded languages HTML and JavaScript into the Java programming language. Usage of our approach results in a system in which every creation of embedded syntax is an explicit action. More specifically, the developer always has to define the exact particularities of the assembled syntax precisely. Therefore, accidental inclusion of adversary-provided semantics is not possible anymore. Furthermore, the only way to assemble embedded syntax is by ELET, which effectively prevents programmers from taking insecure shortcuts. A wide adoption of our proposed techniques would reduce the attack surface of code injection attacks significantly.

Acknowledgements

We wish to thank Erik Meijer for giving us valuable feedback and insight into LINQ.

References

1. American National Standard for Information Technology. ANSI/INCITS 331.1-1999 - Database Languages - SQLJ - Part 1: SQL Routines using the Java (TM) Programming Language. InterNational Committee for Information Technology Standards (formerly NCITS) (September 1999)

2. Halfond, W.G.J., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In: 14th ACM Symposium on the Foundations of Software Engineering, FSE (2006)
3. Hansen, R.: XSS (cross-site scripting) cheat sheet - esp: for filter evasion, http://ha.ckers.org/xss.html (05/05/07)
4. Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th conference on World Wide Web, pp. 40–52. ACM Press, New York (2004)
5. Jalkanen, J.: Jspwiki. [software], http://www.jspwiki.org/
6. Johns, M., Beyerlein, C.: SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In: 22nd ACM Symposium on Applied Computing (SAC 2007) (March 2007)
7. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. In: IEEE Symposium on Security and Privacy (May 2006)
8. Kamkar, S.: Technical explanation of the myspace worm (October 2005), http://namb.la/popular/tech.html (01/10/06)
9. Klein, A.: DOM Based Cross Site Scripting or XSS of the Third Kind (September 2005), http://www.webappsec.org/projects/articles/071105.shtml (05/05/07)
10. Kratzer, J.: Jspwiki multiple vulnerabilitie. Posting to the Bugtraq mailinglist (September 2007), http://seclists.org/bugtraq/2007/Sep/0324.html
11. Livshits, B., Lam, M.S.: Finding security vulnerabilities in java applications using static analysis. In: Proceedings of the 14th USENIX Security Symposium (August 2005)
12. McClure, R.A., Krueger, I.H.: Sql dom: compile time checking of dynamic sql statements. In: Proceedings of the 27th International Conference on Software Engineering (2005)
13. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling Objects, Relations, and XML In the.NET Framework. In: SIGMOD 2006 Industrial Track (2006)
14. Meijer, E., Schulte, W., Bierman, G.: Unifying tables, objects, and documents. In: Declarative Programming in the Context of OO Languages (DP-COOL 2003), vol. 27. John von Neumann Institute of Computing (2003)
15. MSDN. Embedded sql for c, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec_6_epr_01_3m03.asp (27/02/07)
16. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: 20th IFIP International Information Security Conference (May 2005)
17. Ollmann, G.: Second-order code injection. Whitepaper, NGSSoftware Insight Security Research (2004), http://www.ngsconsulting.com/papers/SecondOrderCodeInjection.pdf
18. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 124–145. Springer, Heidelberg (2006)
19. Robertson, W., Vigna, G.: Static Enforcement of Web Application Integrity Through Strong Typing. In: USENIX Security (August 2009)
20. Schneider, J., Yu, R., Dyer, J. (eds.): Ecmascript for xml (e4x) specification. ECMA Standard 357, 2nd edn. (December 2005), http://www.ecma-international.org/publications/standards/Ecma-357.htm

21. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Proceedings of POPL 2006 (January 2006)
22. von Stuppe, S.: Dealing with sql injection (part i) (February 2009), http://sylvanvonstuppe.blogspot.com/2009/02/dealing-with-sql-injection-part-i.html (04/24/09)
23. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 2008. ACM Press, New York (2008)
24. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: 15th USENIX Security Symposium (August 2006)