Towards Practical Prevention of Code Injection Vulnerabilities on the Programming Language Level

Martin Johns

Security in Distributed Systems (SVS)
University of Hamburg, Dept. of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
johns@informatik.uni-hamburg.de

May 28, 2007

Abstract:
A large percentage of today's security problems is caused by code injection vulnerabilities. Many of these vulnerabilities exist because of implicit code generation through string serialization. Based on an analysis of the underlying mechanisms, we propose a general model to outfit modern programming languages with means for explicit and secure code generation. Further, we identify the model's key components: *the language integration*, *the Foreign Language Encapsulation Type*, and *the abstraction layer*. For each of these components we discuss several potential implementation strategies.

Zusammenfassung:
Ein großer Prozentsatz der momentan auftretenden Code-Injection-Verwundbarkeiten existiert aufgrund der üblichen Praxis, dynamisch generierten Code mittels String-Konkatenation zu erzeugen. Basierend auf einer Analyse der grundlegenden Ursachen, die für diese Verwundbarkeitsklasse verantwortlich sind, beschreiben wir ein generelles Modell, das es erlaubt, auf sichere und explizite Art dynamisch Code zu erzeugen. Darauf folgend identifizieren und beschreiben wir die Haupt-Komponenten unseres Ansatzes: die *Language Integration*, den *Foreign Language Encapsulation Type* und den *Abstraction Layer*. Für jede dieser Komponenten diskutieren wir verschiedene Implementierungsstrategien.

1 Introduction

1.1 Outline of this document

Many security vulnerabilities in today's applications arise because these applications are susceptible to code injection attacks. This document proposes first steps towards lowering this threat by enhancing modern programming languages.

In the remainder of this Section we introduce the problem domain and expose the underlying mechanisms that lead to the described vulnerability class. In Section 2 we outline a high level view on our potential language enhancements. In Sections 3 and 4 potential implementation approaches of the proposed method's key components are presented and evaluated. After discussing related work in Section 5, we conclude in Section 6.

1.2 Native and foreign code

Networked applications and especially web applications[1] employ a varying amount of heterogeneous computer languages, such as programming (e.g., Java, PHP, C#), query (e.g., SQL or XPATH), or mark-up languages (e.g., XML or HTML). In the case of web applications, some of these languages are taking effect on the server-side and some in the user's web browser (see Figure 1.1 for an example scenario).

For the remainder of this paper we will use the following naming convention:

- **Native / internal language:** The programming language that was used to program the actual application (e.g., Java or PHP).
- **Foreign / external language:** All other computer languages that are employed by the application.

The application's runtime solely executes the application's native language directly. Foreign code is either passed on to specific interpreters, sent to other hosts, or transmitted to the user's web browser to be processed there. Server-side foreign languages are mostly employed for data management. In this context SQL for interaction with a database and XML for structured data storage in the filesystem are used frequently. Furthermore, for interacting with remote hosts XML-based web-service languages can be found frequently. Finally, on the client side a couple of foreign languages are used to define and implement the application's interface (e.g., HTML, JavaScript, and CSS).

[1] While being suited for any kind of application, our approach aims to counter security vulnerabilities that are notably often found in web applications.

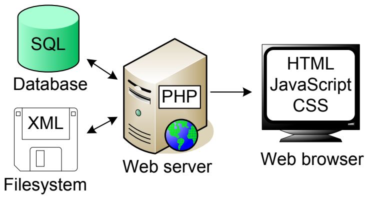


Figure 1.1: Heterogeneous computer languages

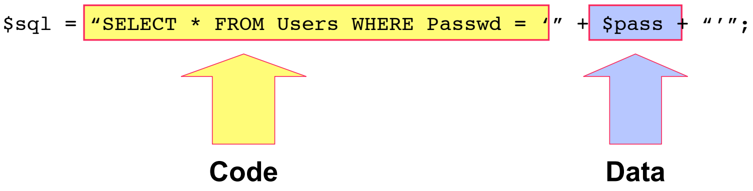
Handling of foreign code: In most cases, an application assembles foreign code exclusively using the native language's String datatype. The native language's interpreter processes all these strings and passes them on to their respective destinations:

```
// foreign HTML code
echo "<a href='http://www.exa.org'>go</a>";
// foreign SQL code
$sql = "SELECT * FROM users";
$con.execute($sql);
```

As all foreign code is handled in form of string values, on a syntactical level the application cannot differentiate strings that contain foreign code from strings that contain general data. Furthermore, during processing, strings that contain foreign code are often combined with data that was obtained on runtime. This way dynamic information can be included in the code, e.g., to outfit a SQL-statement with the ID of an requested dataset. If this dynamically added data is not properly sanitized security problems due to code injection attacks can occur.

1.3 Code injection attacks

Most code injection vulnerabilities arise due to a mismatch in the programmer's intent while assembling the foreign code and the actual interpretation of the code by the external parser. For example, take the following dynamically constructed SQL-statement[2]:



```
$sql = “SELECT * FROM Users WHERE Passwd = ‘” + $pass + “’”;
```

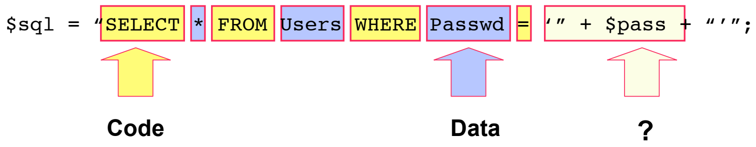
Example 1: The programmer's view on code assembly

The application's programmer probably considered the constant part of the string-assembly to be the code-portion while the added variable was supposed to add dynami-

[2] In the course of this document we will frequently use SQL in examples. Note that the proposed approaches are applicable for any given computer language. The reason for using SQL lies merely in SQL's comparatively expressive syntax that allows the construction of short meaningful code.

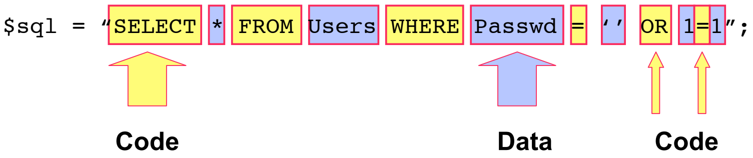
cally data-information to the query.

The database's parser has no knowledge about the programmer's intent. It simply parses the provided string according to the foreign language's grammar:


$sql = "SELECT * FROM Users WHERE Passwd = '" + $pass + "'";
Code
Data
?

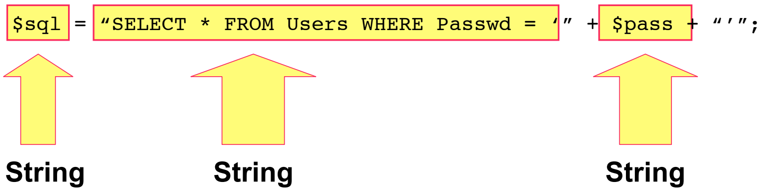
Example 2: The DB's view (generic)

Thus, an attacker can exploit this discord in the respective views of the assembled code by providing data-information that is interpreted by the parser to consist partly of code:



Example 3: The DB's view (code injection)

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 native language does not provide a mechanism to explicitly generate foreign code. For this reason all foreign code is generated implicitly by string-concatenation and -serialization. Thus, the native language has no means to differentiate between user-provided dynamic data and programmer-provided foreign code:


$sql = "SELECT * FROM Users WHERE Passwd = '" + $pass + "'";
String
String
String

Example 4: The native language's view

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 data into a string segment that is passed as foreign code to an external entity, an attacker can succeed in injecting malicious code.

1.3.1 Classes of code injection vulnerabilities

Common classes of code injection vulnerabilities are Cross Site Scripting, SQL Injection, and Remote Command Execution:

Cross Site Scripting (XSS): This class of vulnerabilities subsumes security issues that enable an attacker to inject HTML or JavaScript into a web application's pages. A successful XSS attack can lead to, e.g., the stealing of authentication information, privilege escalation, or disclosure of confidential data.

SQL Injection: In the case of SQL injection, the attacker is able to maliciously manipulate SQL queries that are passed to the application's database. This flaw can lead to, e.g., unauthorized access, data manipulation, or information disclosure.

Remote Command Execution: Sometimes an application dynamically creates code in either its native language or as input to a different server side interpreter (e.g., the shell). If insufficient sanitized, user-provided data is included in this code an attacker may be able to execute arbitrary commands within the application.

2 Achieving code injection resistance

2.1 Lessons learned from the past

A comparison of the security properties of low level languages like C versus modern programming languages like Java yields the observation that a whole class of potential security problems is missing in the latter class: Programs written in such languages are not susceptible to vulnerabilities like Buffer Overflows that arise from errors in a program's memory management. The reason for this is that modern languages do not grant programs direct access to raw memory. Instead a program's memory allocation and usage is abstracted from the actual memory and controlled by internal means of the programming language (See figure 2.1). The lesson learned here is: *A language's security properties are not defined by "what a language can do" but by "what a language cannot do"*. C can write to raw memory. It is therefore subject to Buffer Overflow vulnerabilities. Java cannot write to raw memory. Exploitable Buffer Overflows are therefore impossible.

As described above, code injection attacks are caused by programming constructs that create foreign code with the native language's String type. If we try to apply the lesson we learned from our Java versus C example to code injection flaws, the resulting question would be: *What would a programming language look like that cannot interface directly with external interpreters using the language's String type?*

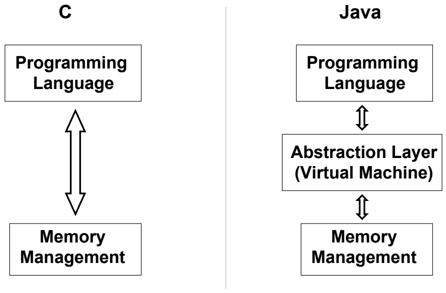


Figure 2.1: Comparing memory management approaches

2.2 High level design considerations

To prevent a programming language to directly interface with external interpreters via its String type, it is crucial to outfit the language with more suitable means to explicitly assemble, encode, and communicate foreign code.

Furthermore, such a language would have to employ an abstraction layer to enable its programs to interface with external entities (see Figure 2.2). For example in the case of web applications the language would e.g. at least require abstracted interfaces for communication with the database and the web browser.

The interaction with the abstraction layer has to be realised using constructs that are an integral part of the native language. To be not susceptible to code injection attacks, it is essential that these constructs provide means for strict separation between data and code.

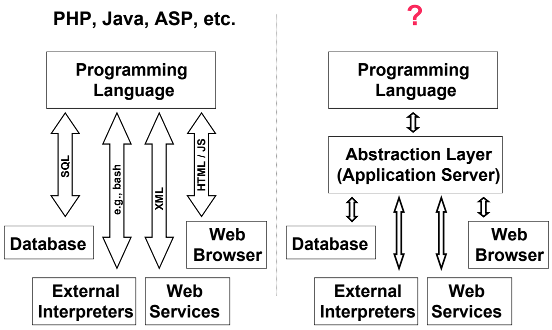


Figure 2.2: Abstracting external language interfaces

2.3 Key components

From the observations detailed in Section 2.2 we can deduct the following key components:

- **Datatype:** We have to introduce a new datatype to the native language that is suitable to assemble/represent foreign code and that guarantees strict separation between data and code according to the programmer's intend.
- **Language integration:** The handling of the newly created datatype and the assembly of foreign language's syntax have to be closely integrated in the native language. Such an integration has to enforce that all creating of foreign code is *explicit* to avoid accidental code creation, e.g., due to implicit string serialization that in turn may lead to code injection vulnerabilities.
- **Abstraction layer:** Furthermore, it is necessary to introduce a separating layer between the application's runtime and the external entities. As the runtime is not

allowed to interact with external interpreters directly anymore, this abstraction mechanism has to handle such communication.

Such an abstraction layer receives the foreign instructions from the application's runtime encapsulated in the newly created datatype. It then translates the provided code information into correct foreign code without being susceptible to injection attacks and passes this code on to the external entity.

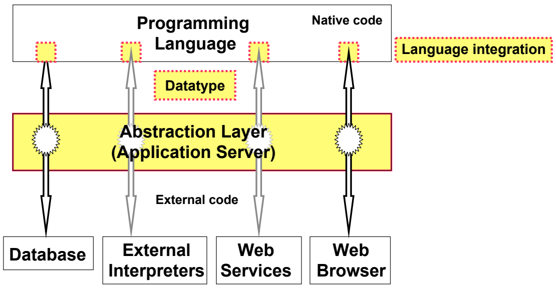


Figure 2.3: Key components of the proposed approach

In the remainder of this document conceptual considerations concerning these components are discussed.

3 Datatype design and language integration

3.1 Design objectives

The specifics how the proposed concepts are realized are crucial to the acceptance by the developer community. An adoption of the new methods is unlikely if such an adoption requires considerable training effort or significant programming obstacles in areas that could be solved conveniently using the current approach. Therefore, in this section we define design objectives that we consider to be fundamental.

Objectives concerning the native language:

Foremost, the proposed concepts should not depend on the specifics of a given native 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 not profoundly change the native language. Only aspects of the native language that directly deal with the assembly of foreign code should be affected.

Objectives concerning the integration of the foreign language:

The specific design of every programming 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, an approach for integrating the foreign syntax in the native language should aim to mimic the foreign language as closely as possible. If the language integration requires profound changes in the foreign syntax it is highly likely that some of the language's original design paradigms are violated.

In addition, the capabilities and flexibility of the String type should be kept. String operations have been proven in practice to be a powerful tool for code assembly. As all assembled foreign code is processed using a specialized datatype, this datatype should therefore e.g., provide means for easy combination of code fragments and capabilities to search and modify the data contents of a given instance.

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

3.2 The Foreign Language Encapsulation Type (FLET)

As described Section 2.3, one of our approach's key components is a datatype which satisfies the following requirements:

- The datatype is tightly integrated in the native language.
- The datatype is capable of encapsulating blocks of foreign code of arbitrary length and complexity while retaining a strict separation between data and code.

We refer to such a datatype as *Foreign Language Encapsulation Type (FLET).* A FLET can be included in the native language either through extending the languages core definition or by implementing the datatype via a programming library. The particular properties of a given FLET are to some degree dependent on the specifics of the foreign syntax that the FLET is supposed to assemble. For this reason we describe in this section only aspects of the datatype that are applicable generally. See Appendix A.1 for exemplified properties of an HTML/JavaScript-FLET.

As the FLET is mainly a mere container to encapsulate instructions for external entities, on an abstract level it is entirely defined by its public API. To allow foreign code assembly this API has to include at least the following methods:

- Methods to create a FLET instance, to add further information to an existing instance, and to combine two instances.
- Separate methods to add either foreign code or data information to an instance. To prevent potential injection attacks, the method to add foreign code cannot rely on arbitrary string-serialization. Instead e.g., a code-keyword based approach could be employed. Depending on the foreign language's properties a FLET might provide a family of method for this purpose.
- Methods to pass the foreign code to the abstraction layer.

Furthermore to emulate certain capabilities of the String-type the FLET should also provide methods to commit the following actions on the data-segments of the FLET's content:

- Search for certain strings using regular expressions.
- Insert, delete or replace specified data-parts after they have been added to an instance.
- Insert further code-information into a data-block (e.g., to create a content filter that adds further HTML markup to a pre-computed page).

3.3 Language integration approaches

The language integration has to outfit the programmer with tools to unambiguously create instructions in the foreign language. In this section we propose three different approaches how foreign syntax could be integrated in the native language. These approaches are then discussed based on the design objectives listed in Section 3.1.

3.3.1 Implementation as an API

A straight forward technique to integrate foreign syntax into a given programming language is to create a high level API that allows the assembly of foreign statements. There are two different design paradigms to create such an API: The API could either emulate the foreign language's syntax (from here on called *syntactic API*) or alternatively recreate the semantics of the language's instructions (*semantic API*).

In the case of a syntactic API the elements of potential foreign instructions are partitioned according to their syntactic function. Possible classes are e.g., language keyword, meta-character, integer-value, or string-value. For each of these classes such an API provides functions to add such an element to a foreign instruction object (see the example below).

Semantic APIs follow the semantics of the language's instructions (examples of this approach would be e.g., the Document Object Model API [10] to create HTML-structures or SQLDom [19] to create SQL-queries). Within this approach the API does not mirror the structure of unparsed source code but the structure of the resulting language object (e.g., the tree structure of a parsed HTML document). Either way, to satisfy the security requirements that are the basis of this paper, the API should provide means for strictly separating executable code from dynamic data.

Example (syntactic API):

```
SQLQuery q = new SQLQuery.addKeyWord("SELECT").addMetaChar("*")
                 .addKeyWord("FROM").addString("Users");
```

Example (semantic API):

```
var newElement = document.createElement(’a’);
newElement.setAttribute(’href’, ’http://www.foo.bar’);
document.addChild(newElement);
```

Advantages: Implementing this approach does not require any changes to the native language or the language's compilation/interpretation process. It is therefore applicable immediately by solely implementing the API.

Disadvantages: The resulting call-structure of a *semantic API* approach differs significantly from the original syntactic structure of the respective foreign language. For this reason the expected training effort is considerable. Furthermore, it has yet to be shown that this approach is applicable for all existing languages. Until now only mark-up languages [10] and query languages [19] have been modeled this way.

In the case of a *syntactic API* the expected training effort of a programmer that is already familiar with the foreign language should be tolerable. However, due to the cumbersome syntax of such an API, creating non-trivial code results in large and overly complicated constructs that are therefore hard to read and maintain.

3.3.2 Extending the native languages grammar

A clean approach towards integrating one computer language into another is to create a combined grammar. This way syntax elements of the foreign code are promoted to first class members of the native language, thus completely eliminating the need to construct source code with the String type.

Example:

```
String UName = "Joe Doe";
SQLQuery q = SELECT * FROM Users WHERE Name = UName;
```

Advantages: Ideally such a solution would not require any syntactical changes in the foreign language. Therefore the objective to closely mimic the foreign syntax is satisfiable and the expected training effort that would be required by an introduction of such a mechanism can be expected to be moderate.

Disadvantages: Implementing such a solution requires profound changes in the native language's compiler or interpreter. Furthermore, the feasibility of this approach is not guaranteed universally. Whether two languages can be combined this way depends on factors like overlapping syntax elements, static vs. dynamic typing, or compiled vs. interpreted execution. It is subject to further research to determine in which cases such an approach towards language integration is possible.

3.3.3 Usage of a pre-processor

By employing a pre-processor the advantages of the two approaches above can be combined without introducing significant additional disadvantages. Instead of directly incorporating the foreign syntax into the native language, an additional mechanism is introduced that transparently translates foreign syntax into appropriate native code. For this procedure a pre-processor that is executing the translation step, and a high level API, representing the foreign code's syntax (see Section 3.3.1), are required. The actual foreign code is integrated in the source code and framed by explicit mark-up signifiers (e.g., $$). Furthermore, to incorporate data-information from the native code into the foreign code statements, the pre-processor has to provide a simple meta-syntax (see example below and Appendix A.2). Before the source code is compiled, the pre-processor translates all foreign code that is framed by according boundaries into the respective API representation.

Example:

```
String UName = "Joe Doe";
SQLQuery q = $$ SELECT * FROM Users WHERE
                Name = $nativeString(UName)$ ORDER BY ID; $$
```

First steps in this direction were realized with SQLJ [6] and Embedded SQL [21], two independently developed mechanisms to combine static SQL statements either with Java or C respectively using a pre-processor (see Section 5.2 for details).

Advantages: By using such a mechanism the foreign language's syntax remains unchanged. Therefore, the expected training effort consists mainly in learning the pre-processors meta-syntax.

Disadvantages: An introduction of such a mechanism requires changes in the compilation process. Before the code can be complied (or interpreted), the pre-processor has to be executed in order to change the foreign code into the native API calls. Therefore, the source code that has been written by the programmer differs from the source code that is processed by the compiler. In the case that a compilation error occurs in a code region that has been altered by the pre-processor, finding and eliminating this programming error may prove difficult. For this reason a wrapped compilation process that post-processes the compiler's messages is recommendable.

4 The abstraction layer

The abstraction layer has the task to serialize an instantiated FLET into a format that is understood by the external entity (e.g., the database engine). In this section we discuss possible alternatives in respect to the abstraction layer's position and potential code serialization strategies.

4.1 Position of the abstraction layer

4.1.1 Integral part of the native language

The abstraction layer could be realized within the means of the native language. Such an implementation would either be done by integrating the layer's functionality in the language's runtime (comparable to Java's memory management) or by implementing a programming library. In either case, the layer provides an interface through which an instantiated FLET is received, serialized to foreign code and communicated to the external entity.

Advantages: Such a realization provides tight integration in the native language. Furthermore, if the layer is implemented via a programming library, the deployment requirements of applications that were written using this mechanism remain unaffected.

Disadvantages: Such a solution is specific for a given native language. This is unfortunate as only a subset of the layer's functionality, mainly the handling of the FLET's internal structure, is specific for a given native language. Other components, like the serialization strategy (see Section 4.2), are independent from the particular properties of the native language.

Additionally if the layer is not implemented as a programming library, changes to either the native language's compiler or runtime are necessary.

4.1.2 Intermediate entity

Secondly, the abstraction layer could be implemented as a detached unit that resides in between the native language's interpreter and the external entity. In this case the native language translates an instantiated FLET into a language independent serialization object, that encodes the foreign instructions while maintaining the strict separation between data and code. This object is then translated by the abstraction layer into actual foreign code.

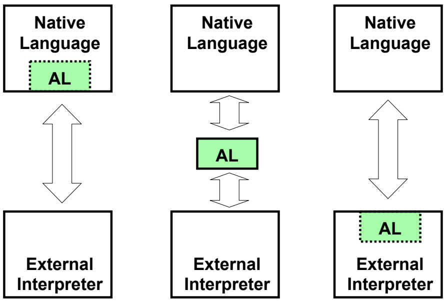


Figure 4.1: Potential positions of the abstraction layer

Advantages: Such an abstraction layer is independent from a specific native language and is thus usable with any language for which a module exists that translates a FLET into the language independent serialization format. Thus, all domain specific knowledge concerning how to create safe foreign instructions has to be implemented only once.

Disadvantages: Realizing the abstraction layer as an intermediate entity adds complexity to the installation process of an application. In addition to the native language's runtime and the external entity also the abstraction layer has to be deployed.

Furthermore, in this scenario an intermediate serialization step is necessary. The FLET is first serialized into the language independent format before this format is translated into foreign code by the abstraction layer, resulting in overhead and potential performance penalties.

4.1.3 Part of the external entity

Finally, the abstraction layer could be directly integrated in the external entity (e.g., into the database's parser). In this case the entity's parsing unit can employ the data/-code information provided by the FLET (which again is communicated as a serialized, language independent object).

Advantages: As discussed in Section 1.3 code injection vulnerabilities occur because of the confusion between data and code portions of dynamically generated foreign instructions. The FLET's internal structure maintains a strict separation between information that was meant by the programmer to be executed and information that was meant to represent data. Additionally, as no step is necessary that translates the FLET's information back into foreign code, potential ambiguities cannot be reintroduced. Thus, by directly using the information concerning data/code separation provided the serialized FLET the external entity's parser can reliably avoid mistakes that lead to code injection vulnerabilities.

In addition to the reliable protection this approach also should provide advantages concerning the performance of applications employing this solution: The serialization step is straight forward and only limited further parsing is necessary, as the FLET already provides information that is structured in a way that is comfortably transferable into a parse-tree.

Disadvantages: Implementing this approach requires profound changes in the external interpreter's interface and parser. While the solutions discussed in Sections 4.1.1 and 4.1.2 can be implemented with programming libraries or additional executables, realizing such an integration demands changing the actual external entity, resulting in high anticipated development costs.

Furthermore, such alteration of the external entity is not always feasible. In certain scenarios the external entity can not be influenced by the deployer of the application. For instance, the operator of a web application has no means to alter the web browsers of the application's users.

Finally such a solution is highly specific for a certain external entity, e.g., one certain database. This property is significant when changes in the deployed technology occur, e.g., when the actual deployed database is exchanged. While a given implementation of the approaches outlined in Sections 4.1.1 and 4.1.2 can be adapted comparatively easy, such a change would require high development cost in the currently discussed solution approach, unless the newly deployed database already supports the FLET's serialization format.

4.2 Foreign code serialization strategy

If the abstraction layer is realized as described in Sections 4.1.1 or 4.1.2 the actual communication with the external entity is still done using a character-based representation of the foreign code. Therefore, the serialization of an instantiated FLET has to be handled with care. Otherwise injection attacks may still be possible.

4.2.1 Disarming potential injection attacks by changing data representation

Some external entities support encoding methods which reliably cause the entity to treat all encoded data as non executable. For example in HTML all characters that are provided in their HTML-encoded version (“`&...;`”) are neither interpreted as HTML nor as JavaScript[1]. If such an encoding is available, all the abstraction layer has to do is to encode the FLET's data information, before passing on the code. Unfortunately, not all external entity types provide such an alternative data representation.

4.2.2 Detecting injection attacks by comparing parse trees

Su and Wassermann have outlined in [26] a method to detect injection attacks. Before passing the foreign code to the external interpreter, the code is parsed twice: Once

[1] Exceptions to this rule, like URL-parameters, exist and have to be treated separately.

the exact code that is supposed to be passed on and once a version of the code in which all dynamically added data is exchanged with dummy data. A difference in the two resulting parse trees is an indicator for an injection attack. Su and Wassermann's approach depends on dynamic data tainting, which is not always feasible and prone to false positives. In our case the tainting step is not necessary as the distinction between data and code is already encoded in the FLET.

Such a solution provides a sound decision whether an injection attack was attempted. Unfortunately, this protection comes with a price: A given foreign code has to be parsed at least three times, twice in the abstraction layer and once in the external entity. Therefore, the resulting overhead especially for large or complex code blocks is expected to be substantial. Furthermore, such a solution would be highly specific for one single external entity type, as the exact foreign parsing process has to be duplicated in the abstraction layer.

4.2.3 Encoding potential "dangerous" entities

Injection attacks can be detected and disarmed by carefully examining the provided data and its execution context. If attacker-provided data attempts to inject code, such an attempt can be detected and disarmed by locally removing or encoding meta-characters that were used to execute the injection attack. This technique closely resembles the current method of output sanitation. In our case there is the significant advantage that the sanitation algorithm has concrete knowledge about the intended nature of the examined code segments.

While being comparatively easy to realize, this approach has to be implemented with great care as otherwise the application might still be vulnerable to sophisticated attacks.

4.2.4 Choosing the appropriate serialization strategy

Whenever possible all dynamically provided data should be re-encoded in a non-executable representation (see Sec. 4.2.1). If the external entity does not provide such an alternative string-representation, the developer should determine if the expected performance overhead of the solution outlined in Section 4.2.2 is compatible with the application's objectives. Only if this is not the case, the "classical" way of output sanitation as described in section 4.2.3 should be implemented.

5 Related Work

5.1 Detecting and mitigating code injection attacks

String Masking: In an earlier work [14] we proposed SMask, an approach towards transparent approximate separation of data and code. To persistently mark legitimate foreign code in string values, SMask applies character masks on certain parts of theses strings before program execution. This way SMask is able to identify code that was injected during the execution process, as injected code is not masked correctly.

Dynamic Taint Analysis: Taint analysis 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 analysis was first introduced by Perl's taint mode [27]. More recent works describe finer grained approaches to dynamic taint propagation. These techniques allow the tracking of untrusted input on the basis of single characters.

In independent concurrent works Nguyen-Tuong et al [22] and Pietraszek and Vanden Berghe [23] proposed fine grained taint propagation to counter various classes of injection attacks. Both approaches require a modification of the interpreter to enhance its string data type. The extended string data type can carry character-level taint information that is preserved by all string operations. A low level integration of the protection mechanism in the native language's interpreter is therefore essential for these approaches to work.

Xu et al [28] propose 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. Therefore, the source code of the interpreter is needed.

Static Taint Analysis: Besides dynamic taint analysis which is done on run-time, there have been proposals for static taint analysis that is solely done by examining the application's source code. Using static source code analysis a data flow graph of the application is generated. Using this graph, the analyzer tries to determine if a data path between the untrusted user input and security sensitive functions exists. Static taint analysis has been described for example by Shankar et al. [25], Huang et al. [11], Livshits and Lam [18], and Jovanovic et al. [15].

Instruction Set Randomization: SQLrand [1] uses instruction set randomization to counter SQL injection attacks. All SQL statements that are included in the protected application are modified to include a randomized component. Between the application and the database a proxy mechanism is introduced that parses every query using the modified instruction set. As the attacker does not know the correct syntax, a code injection attack will result in a parsing error. SQLrand requires the programmer of the application to permanently include the randomized syntax in the application's source code. Therefore SQLrand does not protect legacy applications. Furthermore, as the randomization is static, information leaks like SQL error messages might lead to partial or full disclosure of the randomized instruction set.

Comparison of parse trees: Su and Wassermann [26] describe an approach that employs context free grammars for data validation. Data that is dynamically added to foreign code statements has to fulfill specifically constructed grammars. The approach has been implemented as "SQLCheck" to prevent SQL injection attacks. By tracking dynamically added values through the application's processes SQLCheck can identify untrusted values before the query is passed to the database. These values are parsed by the constructed grammar to validate their correctness. As discussed in Section 4.2.2 their approach is well suited to be employed in our approach's abstraction layer.

5.2 Foreign language integration

Extensive work has been done in the domain of specifically integrating a certain given foreign language into native code (e.g., [2], [16], [3], [9], [7], [17], [4]). Especially SQL and XML-based languages have received a lot of attention. As most of these special purpose integration efforts cannot be extended to arbitrary foreign languages, we only list selected publications in this section.

In order to soften the object-relational impedance mismatch Meijer et al. propose Xen [20], a type system and language extension for C$^{\#}$ that allows native creation and querying of XML-structures. Additionally, Xen promotes a subset of SQL to be first class members of C$^{\#}$. However, the authors still employ string serialization to create XML-data that is passed on to external entities and do not consider the potential security implications of this mechanism.

Russel and Krüger describe SQL DOM [19]. 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 re-generation of the API.

As mentioned in Section 3.3.3 Embedded SQL [21] and SQLJ [6] employ a language pre-processor to enable the embedding of SQL in either C or Java respectively. Unlike our proposed approach these techniques only allow the inclusion of static SQL statements in the source code. The pre-processor creates native code that immediately communicates the SQL code to the database. Thus dynamic assembly and processing of foreign code, as it is provided in our proposed approach via the FLET's interface, is not possible.

6 Conclusion

In this report, we proposed techniques to enhance programming languages with capabilities for secure creation of foreign code.

Based on a generalized model that introduces an abstraction layer between the programming language's runtime and potential external entities, we discussed various alternatives towards the integration of foreign syntax into the native language. The centerpiece of our code-assembly architecture is the FLET (see 3.2), an abstract datatype that allows the creation and processing of foreign code segments while strictly preserving the separation between data and code. This way injection vulnerabilities that are introduced by implicit, string-serialization based code-generation become impossible. We are convinced that an adoption of our proposed techniques would reduce the attack surface of code injection attacks significantly.

The next step in our research will be an exemplified implementation of the preprocessor concept to integrate the languages HTML and JavaScript into Java. This way we aim to prove the feasibility of our approach.

Outlook

During the creation of this document we encountered several topics that should be addressed in the future.

Templates in web applications: Most mature web application frameworks provide templating file-formats to allow the separation of the program's logic from the application's interface (as it can be found in so called "Model, View, Controller"-architectures). Such templating-formats usually combine static HTML-code with well defined insertion points which are filled with dynamic data during execution. Popular examples of such templating-mechanisms are for instance J2EE's *java server pages (jsp)* [12], or Ruby-on-Rail's *rhtml*-format [8].

Depending on the design decisions concerning the FLET's integration in the native language (as outlined in Section 3.3), creating a templating-engine that conforms to our concept's fundamental objectives is not trivial. This holds true especially if the proposed mechanisms are realized in the form of programming libraries (as discussed for example in Sections 3.3.1 (API) or 3.3.3 (preprocessor)) and not as a fundamental addition to the native language's core. In such cases an unsophisticated templating-implementation might reintroduce implicit code-serialization by allowing the creation of foreign code from strings that are stored in files. Consequently, this could provide careless programmers with an insecure shortcut towards foreign code assembly by reading their foreign code from dynamically created files.

However, we consider the existence of a templating mechanism to be essential in the field of web application development. For this reason a potential practical solution has to provide a secure approach towards templating.

Enforcing further coding or security constraints: The FLET encapsulates the foreign code in a partially processed state. Depending on the FLET's actual implementation this state might for example resemble a token stream or an abstract syntax tree. In any case the FLET provides better means towards an automatic "understanding" of the foreign code than the general String-datatype. This could be employed to globally enforce further constraints on the foreign code. For example the well-formedness of dynamically created XML documents could be verified before passing them on to the external entity. Also, depending on the execution context, the FLET could restrict the set of legal code-keywords and APIs to a "safe" subset. For instance, this way third party add-ons/plug-ins to the application can be restricted by a SQL-FLET to use only non-altering database operations like `select`-statements.

Furthermore, in addition to enforcing restrictions on the foreign code, the FLET could also be employed to instrument the foreign code with additional semantics. For example an HTML-FLET could automatically add hidden one-time tokens into HTML forms to avoid CSRF-vulnerabilities [24].

Detection of code injection attempts: As already mentioned in Section 4.2.2 the attack detection method described by Su and Wassermann in [26] is well fitted to be combined with our approach. Su and Wassermann's method is based on comparing parse trees. Every foreign code statement is serialized twice. Once with the dynamic values provided by the application's user and once with dummy values. If the parse trees of the resulting statements differ, the user's data did contain a code injection attack. As the FLET has precise knowledge which components of a given foreign code block are containing data-values, the process of replacing these components with dummy values is straight forward and reliable.

As described in 4.2.2 creating and comparing parse trees in realtime before communicating the foreign code to the external entity may not always be feasible due to performance issues. Furthermore, in many cases the FLET is able to prevent injection attacks reliably by changing the representation of the data-components (see Section 4.2.1) without requiring the identification of malicious data-values in the first place.

However, while not always suitable for *preventing* injection attacks, Su and Wassermann's approach is perfectly fitted for *detecting* attempted attacks. Such a detection mechanism does not require to be applied in real time. The creation and comparison of the parse trees can be done asynchronously after the actual process has taken place. By monitoring potential malicious activities this way, the application's operators can identify source of malicious behaviour like suspicious user accounts or compromised network locations.

A Appendix: Exemplified FLET API and pre-processor metasyntax

As mentioned in Section 6 we are currently working on an implementation that employs the language pre-processor approach (see Section 3.3.3) to incorporate the foreign languages HTML and JavaScript into the Java programming language. For this purpose we did a preliminary definition of an according FLET and a matching pre-processor metasyntax. In this appendix we document our first results in order to clarify the content of sections 3.2 and 3.3.3.

A.1 FLET

As the internals of the FLET are properties of a particular implementation, this section only deals with aspects of the FLET's external interface represented by its API. This API was designed according to the following design paradigms:

- The API is not primarily meant to be used by the programmer. It serves the purpose to provide an interface to the FLET to be used by automatically generated code. As a direct manual usage of this API by the programmer cannot be prevented, the API is specifically designed to prevent the implicit code-serialization of arbitrary string-values.
- The purpose of the API is to inhibit code injection attacks. It does not necessarily mirror the syntactic or semantic meaning of a given element of the foreign language.

A.1.1 Adding HTML code to the FLET

The FLET's core-API consists of the following functions:

- `addText(String text)`: Adds general text to the code. This text represents pure data and contains by definition neither HTML nor JavaScript The abstraction layer is therefore required to treat the text accordingly.
- `openTag(String tagName)`: Adds an HTML element to the encapsulated foreign code. The parameter `tagName` can only contain the name of predefined HTML tags (accordingly to [13]). The tag remains open for potential trailing HTML-attributes which can be added by `addAttribute()`.
- `addAttribute(String attName, String value="")`: Adds an attribute to the preceding open tag. The value of the parameter `attName` has to match one of

the predefined HTML attributes according to [13]. If `addAttribute()` is called outside the context of an open tag, the function parameters are added to the FLET's content as pure data (similar as `addText()` would do).

- `closeTag()`: Closes the last open HTML tag. Remark: `addText()` and `openTag()` also implicitly close preceding open tags.

Example:

```
HTMLFlet h = new HTMLFlet;
h.openTag("a").addAttribute("href", "http://php.net");
h.closeTag();
h.addText("Write better code");
h.openTag("/a").closeTag();
```

A.1.2 Adding JavaScript code to the FLET

JavaScript is an independent self-contained programming language with a syntax that is completely detached from HTML. For this reason, the FLET provides a distinct set of functions to allow adding of JavaScript-code to the encapsulated foreign-code object.

We chose a token-based approach to define the FLET's API in respect to adding JavaScript-code. This means, with the exception of data-values, the elements of the JavaScript's code are added to the FLET in tokenized form. Based on the ECMAScript language definition [5] which standardizes JavaScript's syntax, we differentiate between four distinct element-classes:

1. Keyword-token: Tokens representing elements of JavaScript's set of reserved keywords as defined in [5] Section 7.5.2. For each of these elements the FLET provides a distinct API function. Theses functions adhere to the following syntax convention: `addJS_`*Keyword*`()` where *Keyword* is replaced with the actual keyword (e.g., `addJS_while()` adds the keyword `while` to the code).

2. Identifier-token: Tokens representing programmer defined identifier like variables or function names. Such tokens are added with the function `addJSIdentifier(String id)`. The parameter `id` cannot contain any white-space or illegal punctuation as defined in [5] Section 7.6. Furthermore elements of the keyword-token set as defined above are forbidden.

3. Punctuator-token: Tokens representing legal meta-characters as defined in [5] Section 7.7 (like ";"). For each of these meta-characters the FLET provides a distinct API-function. Theses functions adhere to the following syntax convention: `addJSMetachar_`*Character*`()` where *Character* is replaced by a verbalized representation of the character (e.g., `addJSMetachar_equals()` adds the meta-character "=" to the code).

4. Data-value: String and numeric values are added to the FLET with the functions `addJSStringValue(String val)` and `addJSNumericValue(Double val)` respectively.

Example:

```
HTMLFlet h = new HTMLFlet();
h.openTag("script").closeTag();

// var username = "Peter P. Mary";
h.addJS_var();
h.addJSIdentifier("username");
h.addJSMetachar_equals();
h.addJSMetachar_doublequote();
h.addJSStringValue("Peter P. Mary");
h.addJSMetachar_doublequote();
h.addJSMetachar_semicolon();

h.openTag("/script").closeTag();
```

A.1.3 Combining HTML and JavaScript

JavaScript is included in HTML documents either framed by `script`-tags or as value of an HTML-attribute. The former case can be modeled as exemplified in the listing above. The latter case requires two additional API-elements:

- `startJSAttribute(String attName)`: Adds an HTML-attribute to the preceding open tag. The value of the attribute contains executable JavaScript-code. This code is added by the functions listed in Section A.1.2.
- `endJSAttribute()`: Ends the JavaScript-code value of the preceding opened HTML-attribute.

Example:

```
// <a onclick=’document.location="http://php.net"’>

HTMLFlet h = new HTMLFlet();
h.openTag("a");

h.startJSAttribute("onclick");
h.addJSIdentifier("document");
h.addJSMetachar_dot();
h.addJSIdentifier("location");
h.addJSMetachar_equals();
h.addJSMetachar_doublequote();
h.addJSStringValue("http://php.net");
h.addJSMetachar_doublequote();
h.addJSMetachar_semicolon();
h.endJSAttribute();

h.closeTag();
```

A.2 Metasyntax

To allow unambiguous identification foreign code that is supposed to be handled by the pre-processor, this code is framed by predefined syntactic markers. To avoid potential clashes with the native and foreign language's syntax the actual choice which syntactic markers are employed is variable. In the current case we decided to utilise a metasyntax that is based on the "$"-character as this character does not possess special syntactic meaning neither in Java nor in HTML/JavaScript.

A.2.1 Basic operations

Create a new FLET instance:

```
HTMLFletObj h $=$ <table><tr><td>foo</td></tr> $$
```

Adding further code to a existing FLET instance:

```
h $+$ <tr><td>another table cell</td></tr> $$
```

Combining two FLET instances:

```
HTMLFletObj h1 $=$ <head><title>Homer and Marge $$
HTMLFletObj h2 $=$ sitting on a tree</title></head> $$
h1 $=$ h1 $+$ h2 $$
```

A.2.2 Explicit adding of JavaScript code

The pre-processor can only distinguish JavaScript code from general text through the HTML-context of the particular code fragment. Such a context would be either framing `<script>`-tags or an attribute-definition that expects JavaScript code inside its value (e.g., an event handler). If the pre-processor should create JavaScript code outside of such a context the programmer has to communicate his intention explicitly. Otherwise the text would be added to the FLET as non-executable data.

```
h $+JS$ document.write("Hello World"); $$
```

A.2.3 Combining foreign code with native datatypes

To allow the dynamic creation of foreign code the meta-syntax contains means to add data-information provided by the native language. In particular the meta-functions `$addString()$`, `$addInteger()$` and `$addDouble()$` are provided to allow the inclusion of Java's basic datatypes.

```
// Add dynamic string information to foreign code statement
String email = req.getParameter("email");
HTML h $=$ <a href="mailto:$addString(email)$">Email</a> $$

// Add dynamic integer information to foreign code statement
int id = get_some_reference_ID(parameter1, parameter2);
h $+$ <a href="page.jsp?id=$addInteger(id)$">link</a> $$
```

Bibliography

[1] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X, October 2002.

[3] R. Connor, D. Lievens, F. Simeoni, S. Neely, and G. Russell. Projector: a partially typed language for querying xml. In *Programming Language Technologies for XML (PLAN-X 2002)*, 2002.

[4] William R. Cook and Siddhartha Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. of the International Conference on Software Engineering (ICSE 2005)*, pages 97–106, 2005.

[5] ECMA. Ecmascript language specification, 3rd edition. Standard ECMA-262, `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, December 1999.

[6] 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.

[7] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[8] David Heinemeier Hansson. Ruby on rails documentation. [online], `http://www.rubyonrails.org/docs`, (05/18/07), 2007.

[9] Falk Hartmann. An architecture for an xml-template engine enabling safe authoring. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 502–507, 2006.

[10] Philippe Le Hegaret, Ray Whitmer, and Lauren Wood. Document object model (dom). W3C recommendation, `http://www.w3.org/DOM/`, January 2005.

[11] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[12] Sun Microsystems Inc. Javaserver pages technology. [online], `http://java.sun.com/products/jsp/`, (05/18/07), 2007.

[13] Ian Jacobs, Arnaud Le Hors, and David Raggett. Html 4.01 specification. W3C recommendation, `http://www.w3.org/TR/1999/REC-html401-19991224`, November 1999.

[14] Martin Johns and Christian Beyerlein. Smask: Preventing injection attacks in web applications by approximating automatic data/code separation. In *22nd ACM Symposium on Applied Computing (SAC 2007), Security Track*, March 2007.

[15] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, May 2006.

[16] Martin Kempa and Volker Linnemann. On xml objects. In *Programming Language Technologies for XML (PLAN-X 2002)*, 2002.

[17] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[18] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications using static analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[19] R. A. McClure and I. H. Krueger. Sql dom: compile time checking of dynamic sql statements. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[20] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects, and documents. In *Declarative Programming in the Context of OO Languages (DP-COOL '03)*, volume 27. John von Neumann Institute of Computing, 2003.

[21] MSDN. Embedded sql for c. website, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec_6_epr_01_3m03.asp`, (27/02/07).

[22] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.

[23] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[24] Thomas Schreiber. Session riding - a widespread vulnerability in today's web applications. Whitepaper, SecureNet GmbH, `http://www.securenet.de/papers/Session_Riding.pdf`, December 2004.

[25] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[26] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of POPL'06*, January 2006.

[27] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[28] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, August 2006.