SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation*

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

Christian Beyerlein
Security in Distributed Systems (SVS)
University of Hamburg, Dept. of Informatics
Vogt-Koelln-Str. 30, D-22527 Hamburg
9beyerle@informatik.uni-hamburg.de

ABSTRACT

Web applications employ a heterogeneous set of programming languages: the language that was used to write the application's logic and several supporting languages. Supporting languages are e.g., server-side languages for data management like SQL and client-side interface languages such as HTML and JavaScript. These languages are handled as string values by the application's logic. Therefore, no syntactic means exists to differentiate between executable code and generic data. This circumstance is the root of most code injection vulnerabilities: Attackers succeed in providing malicious data that is executed by the application as code. In this paper we introduce SMask, a novel approach towards approximating data/code separation. By using string masking to persistently mark legitimate code in string values, SMask is able to identify code that was injected during the processing of an http request. SMask works transparently to the application and is implementable either by integration in the application server or by source-to-source translation using code instrumentation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Preprocessors, Interpreters*

General Terms

Security

Keywords

Code Injection, Web Application

*This work was supported by the German Ministry of Economics and Technology (BMWi) as part of the project "secologic", www.secologic.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'07 March 11-15, 2007, Seoul, Korea
Copyright 2007 ACM 1-59593-480-4 /07/0003 ...$5.00.

1. INTRODUCTION

Nowadays web applications are ubiquitous. They are not confined anymore to colorful communication, advertising and online banking. Modern web applications are now used for critical applications like project management, enterprise resource planing, or supply chain management. Unfortunately web applications are prone to various classes of code injection attacks. In the first six months of 2006 alone more than 550 cross site scripting and over 350 SQL injection vulnerabilities were posted to the BugTraq mailing list. With the ever growing importance of web applications the impact of these vulnerabilities grew accordingly from being mere annoyances to actual severe threats.

In this paper we discuss the underlying mechanisms of code injection attacks and propose SMask, a protection mechanism based on automatic separation between data and code.

1.1 Native and foreign code

Web applications employ a varying amount of heterogeneous computer languages, some taking effect on the web server and some in the user's web browser (see Figure 1 for an example scenario). Most important on the server side is the language that was used to program the actual web application's logic (e.g., PHP or Java). From now on we refer to this language as the application's *native* language. All other languages found in a given web application are denoted as *foreign*.

Web applications are usually written in interpreted scripting languages like PHP and Perl or in languages like Java that compile into byte code. In the latter case the byte code is interpreted by a virtual machine. Languages that compile into binary code are rarely used to write web applications. While our approach is applicable for either variation, in this paper we only refer to interpreted languages. The web server itself solely executes the application's native language directly using the language's interpreter. Foreign code is either passed on to specific interpreters or sent to the user's web browser to be executed 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. On the client side a couple of foreign languages are used to define and implement the application's interface (e.g., HTML, JavaScript, CSS).

Location of foreign code: Web applications can include foreign code from different origins. Often foreign code is directly part of the application's source code. In this case,

the foreign code is kept in static string constants. Furthermore, the application can obtain foreign code on runtime from external data sources like databases or the filesystem, an example being predefined HTML templates. In these cases the interpreter reads the foreign code into string variables. The native language's interpreter processes all these strings and passes them to their respective destinations (see Listing 1).

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

Listing 1: Examples of embedded foreign code

As all foreign code is handled in form of string values, on a syntactical level the web application has no means to differentiate strings that contain foreign code from strings that contain general data. Furthermore, during the processing of an http request 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 read a certain user's dataset from the database. If this dynamically added data is not properly sanitized security problems can arise.

1.2 Code Injection Attacks

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 if a flaw in the application's logic allows an execution of this data, an attacker can succeed in injecting malicious code. 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. See Section 2.1 for an example of this vulnerability class.

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 on the web server.

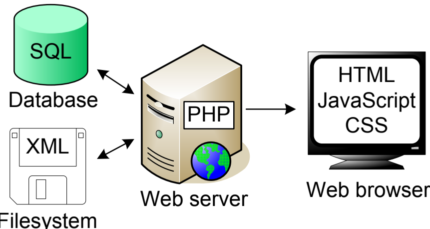

Figure 1: Heterogeneous programming languages

2. DATA/CODE SEPARATION

As described above many insecurities in today's web applications are rooted in confusing data and code. By exploiting this common flaw, an attacker is able to inject code or code fragments into the application. As shown above foreign code in web applications is handled as content of strings. Our approach aims to automatically deduct if a given string constant represents foreign code or generic data, thus providing an approach to data and code separation.

In general it is undecidable if a given string constant will be executed as foreign code in a given environment. As a full formal proof of this claim is out of the scope of this paper, we provide a sketch of this proof which is a variant of Rice's theorem [13]: Suppose there is an algorithm $A(C, I)$ that returns `true` if given an input I a program C uses its first string constant for code execution. Then we could build a program P that includes A, which contradicts the initial hypothesis:

```
P(I) = {
    string s = 'rm -rf /'
    if (A(I,I) == false)
        execute(s);
    else
        print(s);
}
```

If we now run $P(P)$, a call to P with its own source code as input, a contradiction is triggered: If A decides that P will not execute s, s gets executed by P and if A decides that s will be executed by P, P merely prints s and exits.

As the general problem to decide if a given string constant will be executed in a given situation (and therefore contains code) is undecidable, we have to approximate a solution. We choose an over-approximation approach. In this context over-approximation results in an algorithm that identifies all strings that will be executed. However such an over-approximating algorithm may also falsely classify some general data to be code.

2.1 String Masking

In this section we outline our approach towards approximative data/code separation. As web applications handle foreign code and generic data in strings, there is no syntactic means to differentiate between these two classes during program execution. For this reason we propose *string masking*, a method that enables our technique to syntacticly mark foreign code in strings without changing the native language's string type.

General approach: Our proposed method is based on the following assumption: All general semantics of foreign code are completely defined in the application's source code and static data sources. Dynamically obtained data, like user input, is only used to add data values to the predefined code. In other words: User input never contains actual code. Obviously, there are exceptions to this rule, see Section 2.3 for details.

Legal and illegal code: This assumption leads us to the following definition: All foreign code that is part of the application before the processing of an http request is *legal*. More precisely: Legal foreign code is either part of the application's source code or contained in specific trusted data

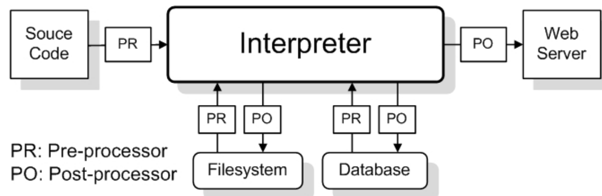

Figure 2: Schematic view of the processes

sources like database tables or files. Only this foreign code is allowed to be executed or included in the application's web pages. Accordingly, all foreign code that is added dynamically to the application in the course of processing data is potentially malicious and should be neutralized. This code is from here on denoted as *illegal* code. Please note an important distinction in this matter: This definition does solely apply to actual code and not to data values that have been added to preexisting legal code.

Example 1 (illegal code): An application contains the following predefined SQL instruction:

```
$sql = "SELECT * FROM USERS WHERE ID = $id
        and PASS = '$pw'";
```

The variables `$id` and `$pw` are placeholders for dynamic data values representing the ID and password of a user. If the application uses the values `42` and `foo` to complete the SQL statement, the resulting code looks like this:

```
"SELECT * FROM USERS WHERE ID = 42
 and PASS = 'foo'"
```

The foreign code represented by this string is still legal, as only data values have been added on runtime. But if an attacker can figure out a way to pass arbitrary values to either variable, the application is vulnerable to SQL-injection. In this case the attacker can pass the string "`bar' OR '1' = '1`" as value for `$pw` to bypass the application's authentication mechanism. Then the resulting SQL string would look like the following:

```
"SELECT * FROM USERS WHERE ID = 42
 and PASS = 'bar' OR '1' = '1'"
```

Now the foreign code in the string is not legal anylonger. Besides the data values also semantic elements have been added dynamically. The signifiers "`OR`" and "`=`" do not classify as data values but are instead language elements that change the code's actual logic.

Note: In the following paragraphs we describe our technique solely in respect to JavaScript and countering Cross Site Scripting attacks. This is done to avoid unnecessary complex descriptions. The technique itself is not limited to XSS but applicable to counter various injection attacks. See Section 3.4 for details on that matter.

Security Pre- and Post-processor: As shown in Section 1.1, foreign code is embedded in web applications either as part of string constants or in kept in external data sources. We introduce a pre- and a post-processor for illegal code detection. For every string constant the preprocessor enforces a syntactic separation between data and code by masking certain parts of the string. After the processing of an http request, the post-processor detects and neutralizes

Language	Keywords
HTML	– All reserved HTML tag names with more than two characters e.g., html, head, body, script – All predefined HTML attributes e.g., href, src, onload, onmouseover
JavaScript	– All reserved JavaScript words e.g., for, while, try, eval – Selected DOM signifiers e.g., document, children, getElementById – Names of global JavaScript objects e.g., window, location

Table 1: Used keywords

illegal code. Furthermore, the post-processor removes the masks that were applied by the pre-processor.

Masking legal code: Before processing the http request, all foreign code is masked to enable deterministic identification of legal code in the course of processing the request. To find legal code in the application's data a set of keywords is used. The pre-processor examines every string constant that is part of the application's source code whether it contains any of the specified keywords. If such a keyword was found, the keyword is replaced with a code mask. This mask is a per-request random token representing the found keyword. If this keyword is found more than once, it is always replaced by the same token. The masking is done on the first initialization of the string constant. Furthermore, all server side input, that has been specified to contain legal code, is filtered accordingly: All found keywords are replaced by per-request code masks. This way it is ensured that all legal foreign code is masked before it enters the application. The pre-processor stores the pairs consisting of the original keywords and the per-request masks in a table to enable the reverse transition later on.

Keywords: A proper set of keywords has to be chosen to counter the projected attacks. By definition a keyword is a single entity. In the source code it is always framed by non-alphabetic characters. As mentioned above XSS attacks are actually HTML/JavaScript injection attacks. To counter these attacks the keyword list consists of reserved HTML signifier, HTML attributes, reserved JavaScript words and names of global JavaScript/DOM objects (see table 1).

Detecting and encoding illegal code: The post-processor searches for foreign code before the resulting HTML data is sent to the browser. This is done using the same keyword list as the pre-processor. As all legal code has been priorly masked, all foreign code that can be found is either illegal or not code at all. In the latter case the suspicious code is actually textual data that contains code keywords, e.g., a forum post discussing JavaScript issues. The post-processor encodes all found keywords using HTML entities.

Reversing the code masking: After all illegal code has been encoded, the final processing step is applied. The post-processor examines the HTML code in the web server's output buffer and uses the mapping between keywords and random tokens to reverse the pre-processors actions. All previously masked keywords are restored before sending the http response to the user's browser.

Example 2 (masking): An application contains the instruction:

```
echo "<a href='user.php'>You</a>
      searched for $term.";
```

If the variable `$term` is not sanitized by the application, this would be a classical example for a cross site scripting vulnerability. The pre-processor identifies the keyword `href` and applies an one-time mask:

```
echo "<a _x2sgth_='user.pl'>You</a>
      searched for $term.";
```

If a malicious user would try to exploit the XSS vulnerability by passing JavaScript code to `$term`, the output buffer may look like:

```
<a _x2sgth_='user.pl'>You</a>
searched for <script>var a=...
```

Before sending the resulting web page to the web browser, the post-processor scans the output buffer for keywords again. Thus he finds the injected code. This code is encoded using HTML entities:

```
<a _x2sgth_='user.pl'>You</a>
searched for <&#115;&#099;&#114;...
```

This encoding disarms the illegal code, as it is no longer recognized as HTML or JavaScript by the web browser, thus neutralizing it effectively. After all illegal code has been encoded, the post-processor removes the code mask:

```
<a href='user.pl'>You</a>
searched for <&#115;&#099;&#114;...
```

The resulting HTML document is included in the http response.

Communication with outside data sources: As mentioned above and shown in Figure 2 the pre- and post-processor also handle string data communication with outside entities like the file system or databases. Legal code in incoming strings is masked and string masks are removed from outgoing data. Furthermore, the post-processor removes present string masks from identifiers like file- or table-names before accessing the outside data source.

2.2 False positives and false negatives

In order to assess the provided protection of the proposed mechanism, we have to consider cases where the described processes fail. A failure could either be a *false positive* or a *false negative*. With a *false positive* we describe the false identification of harmless data as illegal code. With a *false negative* we describe the case, if one of the processors fails to detect foreign code. The potential consequences of these two failure classes are fundamentally different: A false positive might lead to problems in the further execution of the web application. A false negative may enable a successful code injection attack. We examine each potential failure case to establish the probability of its occurrence and its potential impact.

Pre-processor false positives: A false positive of the pre-processor occurs if a string constant contains a foreign code keyword outside the context of actual foreign code. This might happen for different reasons:

The keyword could be part of textual data. For example the DOM tree signifier "document" may be used in a text about old manuscripts. This false positive is neither noticeable nor problematic, as before the textual data leaves the scope of the native code, the post-processor removes the code mask and restores the original text.

Furthermore, the found keyword could be used to conduct operations on the produced webpage. For example sophisticated web applications apply filters on HTML code before passing it to the browser. These filters may use strings for search operations on the HTML code. Again, as the mapping between keywords and code masks is static, such operations can function as intended. Our proposed technique does not change the syntactic structure represented in the application's strings. It just replaces certain string constants, the keywords, with other string constants, the code masks.

Finally, the identified keyword could be in fact a textual identifier used in one of the application's native language constructs. Some programming languages use strings to access data stored in containers like hashtables. If the hashtable's identifiers are only used inside the application source code such a false positive does not yield any problematic consequences. The mapping between the keyword and the code mask is static for the complete processing of an http request. Thus, the deterministic referencing of information is unhindered. But if the hashtable's identifiers are derived from an outside entity like a database table or the actual http request, the access to the hashtable's information may be hindered. In this case the pre-processor has to be adapted slightly to avoid the false positive.

Post-processor false positives: As it is the case with the pre-processor, a false positive of the post-processor occurs when a keyword is found in a non-code context. In this case the keyword is either part of the webpage's text or a value of a dynamically generated HTML attribute. The former case does not pose any problems. The post-processor encodes the found keyword in HTML entities which are displayable by the web browser.

To discuss the latter case we have to differentiate between different attribute types. We can divide HTML attributes in six classes based on the value type they accept (see table 2 for details). HTML attributes accept either numerical values, identifiers, predefined textual data, JavaScript, URLs, or variable textual data. Three of these classes are safe in this context as their values cannot cause false positives: Numerical values cannot include foreign code keywords, identifiers are not derived from user input, and the list of predefined attribute values does not intersect with the used list of foreign code keywords. Furthermore, by definition there cannot be a false positive concerning JavaScript, as our countermeasure explicitly aims at detecting and neutralizing rogue script code. False positives in URL-attributes lead to URLs that are partly encoded in HTML entities. As modern web browser interpret encoded URLs correctly, such an incidence does not disturb the web application. This leaves false positives in attributes that accept variable textual data. Within this class we have not encountered problems caused by false positives. Commonly used attributes from this class are either never parsed by the web browser (e.g., `rel`) or work transparent with HTML encoded values (e.g., `alt` or `value`). But as such attributes are sometimes used for purposes that are not covered in the HTML specification [5], there may be rare problematic scenarios.

Pre-processor false negative: As the encoding of the server side data is controlled and deterministic, such a false negative can only happen if the used list of keywords is incomplete.

Value type	Examples
Numerical value	width, height
Identifier	class, name, id
Predefined textual data	align, type, method
JavaScript	onload, onclick, onfocus
URL	href, src
Variable textual data	alt, value, rel

Table 2: Value types of HMTL attributes

Post-processor false negatives: The correct detection of illegal code depends on the ability of the post-processor to find specific keywords in the output buffer. Attackers are known to employ various input encoding techniques to evade filter mechanism [15]. The basic idea behind most of these evading techniques is to create HTML code that does not comply with the general grammar of HTML [5] but is still recognized by the web browser. The reason which enables this approach is that HTML parsers are known to employ a rather forgiving parsing process. This behavior enables web browsers to render web pages that contain faulty HTML code. The post-processor has to take all known evading techniques into consideration to be able to detect such obfuscated HTML tokens. But as it is undocumented which syntax errors in HTML code the diverse browsers accept, there might still exist undiscovered evading techniques. Because of this special characteristic of HTML parsers an attacker might be able to craft data that contains HTML code which is not detectable by the post-processor. This way the attacker may under certain conditions succeed to include a rogue HTML tag into the resulting webpage. If such a previously undiscovered evading technique is discovered the mechanisms can be adapted easily, as the post-processor is a single central component. Nonetheless, the inclusion of a functioning JavaScript is not feasible: Other parsers are strict in which syntax they accept or reject. For this reason the techniques described in [15] do not apply to programming languages like JavaScript or SQL. For a successful injection attack the complete injected code has to evade the detection process. A single detected and encoded code fragment causes the attacked interpreter's parser to recognize a syntax error and abort the execution of the injected code.

2.3 Allowing Foreign Code

There are legitimate scenarios in which a web application needs to generate foreign code from dynamically obtained data, e.g., discussion forums that permit a subset of HTML, web based database frontends that provide an SQL shell or a content management system that allows its administrator to include JavaScript in the system's pages. To enable such dynamic generation of foreign code, we introduce URL based policies. Such a policy whitelists single foreign code keywords. Before encoding illegal code the post-processor checks if one of the application's policies matches the request's URL. If this is the case, the post-processor skips the encoding of the keywords that are specified in the matching policy.

Also, these policies are used by the pre-processor to identify trusted data-sources. String values that are obtained from such sources are masked as if the strings were part of the application's source code. As the pre- and post-processor are single application-global entities, these policies provide a central mechanism to control certain security properties of the application, e.g., the particulars of user provided HTML.

Examples: A policy to allow a weblog's visitors to add images to their comments would look like this:

```
<policy unit="post-processor">
    <url>/blog/comments.php</url>
    <keyword>img</keyword>
    <keyword>src</keyword>
</policy>
```

Accordingly, a policy specifing an external data source as trusted to contain legal HTML code would look like this:

```
<policy unit="pre-processor">
    <file>/templates/*</file>
    <keyword>img</keyword>
    ...
</policy>
```

2.4 Implementation approaches

There are two distinct approaches to implement the proposed methods. Either the described measures can be directly integrated in the native language's interpreter/compiler or they can be implemented by instrumenting the source code. The details of a direct integration are very dependent on the specific language. Therefore, we omit a general description of this approach for brevity reasons. See Section 3.1 for a concrete example.

Code instrumentation is an automatic source-to-source transformation that wraps certain functions with calls to either the pre- or the post-processor. If code instrumentation is used, the source code of the whole application is modified before passing it to the interpreter. The actions of the pre- and post-processor are implemented as regular functions and added to the application's code. All static strings are wrapped by the pre-processor function. Furthermore, all function calls that retrieve string values from external data sources are wrapped as well. Accordingly the post-processor wraps all function calls, that cause string data to leave the system. The application's source code has to be instrumented only once. The modified source code can be stored permanently and serve as the application's actual code base.

Example: Before instrumentation:

```
// static string constants
$code = "<script>...</script>";
// accessing external string constants
$data = fread($file, 100);
// writing string data
fwrite($file, $data);
```

After instrumentation:

```
// static string constants
$code = __smPrepro("<script>...</script>");
// accessing external string constants
$data = __smPrepro(fread($file, 100));
// writing string data
fwrite($file, __smPostpro($data));
```

Implementing the final post-processing of the produced HTML code with code instrumentation may not always be possible. In this case two alternative solution exist to realize the post-processor without modifying the actual interpreter: If the language provides an output buffering mechanism, which collects the complete HTML code before passing it to the

web server, this mechanism can be employed to implement the final post-processing. However some languages do not provide such a buffering mechanism. In this situation, a proxy mechanism between the language's interpreter and the web server has to be introduced.

2.5 Generality the approach

As mentioned above our approach is neither limited to countering cross site scripting nor to web applications. In this section we give examples for further possible deployments:

SQL Injection and Remote Command Execution: Applying our proposed method to counter other code injection attacks is in general a matter of extending the list of keywords and adapting the policies. Only the post-processor's method to neutralize potential malicious code is dependent on the nature of the protected interpreter. To avoid unwanted consequences of false positives, we use string encoding methods whenever possible. For example SQL dialects usually provide the function `char()` that takes numerical values and translates them to the corresponding characters. This function can be used to disarm potential SQL Injection attempts in the same fashion as we used HTML encoding to counter XSS attacks.

Directory traversal and shell injection: These two classes of attacks are based on the injection of meta characters like `..`, `|`, or `&&`. As the semantics of these signifiers are on the same abstraction level as code keywords, our approach is also applicable to counter these attack classes.

3. DISCUSSION

3.1 Practical implementation using PHP

For a practical implementation of our concept, we chose a direct integration into an interpreter. We decided in favor of the direct integration over a source-to-source instrumentation approach, as we anticipated such an implementation to be easy integrable in existing setups, thus encouraging SMask's usage. We implemented SMask as an PHP5 extension [16]. PHP extensions are powerful libraries that are plugged directly into the PHP interpreter. They can access global data structures, pre-process an http request's data, apply operations on PHP's output buffer, introduce new functions to PHP, and modify the semantics of existing ones. Additionally, we added four lines of code to the source code of the PHP interpreter. This had to be done to enable SMask's integration in PHP's parser.

To mask code in static string constants that are part of the applications source code, SMask injects a hook in the PHP parsing process. This hook causes PHP's lexer to pass the lexical tokens to SMask's pre-processor. The pre-processor examines these tokens whether they represent one of PHP's different string constants. If this is the case the token's data is masked according to the list of keywords. Subsequently the token is passed on to the actual parser. In order to intercept communication with external data sources like the filesystem or a database, our extension redirects calls to the respective API functions through either the pre- or the post-processor. Furthermore, PHP communicates request-global data like the POST and GET parameters via hashtables. For this reason the extension implements a SAPI input filter which examines these specific tables whether they contain keys that match one of the keywords. If such a key is found, it is masked to allow unhindered access to the hashtable's values. Finally, our extension registers a handler for PHP's output buffer. This handler applies the post-processing operations on the http response's body.

3.2 Evaluation

The practical evaluation of our approach was twofold. On the one hand we examined if our implementation is compatible with existing applications, on the other hand we assured that our concept indeed provides the desired protection.

Evaluation of compatibility: At first we examined if execution problems occur when existing PHP applications are run on a PHP system that uses our SMask extension. For this reason we installed several popular open source PHP applications. See Table 3 for details. All tested web applications worked as expected without any modifications.

Application	Version	Vulnerability
PHPMyAdmin	2.8.0.3	[none]
PHPNuke	7.8	XSS in search module [1]
PHPBB	2.0.16	XSS in nested tags [12]
Wordpress	2.0.4	[none]
Tikiwiki	1.9.3.1	Multiple XSS issues [2]

Table 3: List of tested PHP applications

Evaluation of protection: In order to verify that our technique indeed prevents XSS attacks, two testing approaches were applied. For one we tested known XSS attack methods against a self written test script and secondly we examined vulnerable versions of popular applications.

Our test script solely consists of a simple echoing function, that writes all user input directly unfiltered in an HTML page. Using the XSS attacks listed in [15], two different policies were evaluated: One policy that prohibits all user-supplied code and one policy that allows a typical HTML subset, thus permitting the dynamic inclusion of basic text-formating and usage of hyperlinks. Both policies prevented our XSS attacks.

Then we installed three PHP applications with public disclosed XSS flaws (see Table 3). Executing the attack vectors that were documented in the respective advisories, we verified that SMask successfully prevented the exploit.

3.3 Protection

If an attacker injects correctly masked code, this code is translated to working foreign code by the post-processor. Therefore, the measure's effectiveness in protecting against injection attacks depends on the ability of an attacker to guess correct code masks. As every keyword is masked differently, the attacker has to guess the individual masks for all keywords used in his attack. For this reason the success probability of such an attack shrinks with the number of keywords used in the attack. To estimate a lower bound for this probability we have to look at attack vectors with as few keywords as possible. For example the following string represents the smallest XSS attack vector that is able to conduct a meaningful attack:

```
<script src="http://a.org/a.js"></script>
```

This vector contains only two keywords: `script` and `src`. If the processors employs code-masks of length eight over an

alphabet of 62 symbols (numbers and characters in upper and lower case), the probability of a successful attack is:

$$P_{success} = \frac{1}{62^8(62^8-1)}$$

In the case that the keyword `src` is permitted, e.g., by a site that allows its users to post images, the probability is:

$$P_{success} = \frac{1}{62^8}$$

These probabilities are constant over a series of attacks, as the code masks change for every single http request. For the same reason hypothetical information leaks pose no security problem.

3.4 Future work

Our proposed approach to determine if a string contains code is rather coarse. While the described method allows efficient implementation for on-the-fly checking of string values, it produces a certain number of false positives. Efficient on-the-fly checking is an important property only for implementations that are directly included in the language's interpreter. However, source-to-source code instrumentation can be mostly pre-calculated and the resulting code can be stored for actual usage. If this approach is chosen, more sophisticated methods for code detection are applicable. We plan on investigating algorithms that employ the application's control flow graph to improve our approximation.

Furthermore, advanced source-to-source translation can also improve the SMask's performance. As stated in the last paragraph, most of pre-processing has to be done only once, for example the decision step whether a static string constant qualifies as potential code:

```
// Naive approach:
string $s = __smPrepro("<body> Hello");
// Improved approach:
string $s = __smMask("<body>") + " Hello";
```

Finally, there is room for improvement in the field of policies. Instead of solely relying on keyword matching, we can use more sophisticated techniques to determine if dynamically added foreign code is legal or illegal. Such techniques can e.g., take relationships between single code fragments into consideration.

3.5 Related work

In the last decade extensive research concerning the prevention of code injection attacks has been conducted. In this section we only discuss works that describe protection mechanisms for web applications. We discuss the approaches in general and if applicable, how they compare to our work.

Manual approaches: To counter injection attacks, applications have to treat untrusted data with caution. For this reason input validation and output sanitization are employed. Input validation procedures check if incoming data matches predefined specifications. Output sanitization takes effect before HTML data is sent to the web browser, aiming to remove malicious content. Both tasks depend on various factors, e.g., the expected data types, the role of the connected user, or the actual execution context. Therefore, a single centralized, well audited implementation is seldom possible. Instead web applications frequently have to rely on several protection mechanisms that are scattered through the code. For this reason manual protection against injection attacks is a complex and error prone task. For example, in 2005 a XSS input filter was added to the PHPNuke content management system that still was vulnerable against numerous known XSS attack vectors [1].

Countering SQL Injections: SQLrand [3] 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. In comparison our approach works transparently with legacy applications and employs different code masks for every request, thus rendering information leaks harmless.

Su and Wassermann [18] 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.

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 [8]. 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. While coming from a fundamentally different direction this approach is potentially as powerful as ours: Before passing code to external interpreters all keywords that contain tainted characters are rejected or encoded.

In independent concurrent works Nguyen-Tuong et al [10] and Pietraszek and Vanden Berghe [11] 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. In comparison our technique can be implemented using code instrumentation as described in Section 2.4 and requires therefore no necessary alteration of the interpreter or application server. Xu et al [19] 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 while our technique can also be employed for closed source languages using the code instrumentation approach.

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 for web application has been described by Huang et al [4], Livshits and Lam [9], and Jovanovic et al [6].

Web application firewalls: The term web application firewall describes applications that are positioned between the network and the web server. Using a specific ruleset, incoming data is modified or removed to counter injection attacks. In Scott and Sharp's [17] proposal the firewall's ruleset is defined in a specialized security policy description language. According to this ruleset incoming user data (via POST, GET and cookie values) is sanitized. Only requests to URLs for which policies have been defined are passed to the web server. The Sanctum AppShield Firewall is another web application firewall [7]. AppShield executes default filter operations on all user provided data in order to remove potential XSS attacks. AppShield requires no application specific configuration which makes it easy to install but less powerful. Furthermore, Mod_security [14] is an open source web application firewall specific for the Apache web server that allows detailed analysis and modification of incoming http requests. Web application firewalls can only substitute input validation mechanisms, as they do not possess any knowledge about the application's internals. The provided protection is therefore seldom complete.

4. CONCLUSION

In this paper we proposed SMask, a novel approximation to automatic data/code separation for countering injection attacks. SMask employs string masking to introduce a syntactical means which enables the web application to differentiate between legitimate and injected code. This way a variety of code injection attacks can be prevented.

SMask can either be implemented by integration in the native language's interpreter or by automatic source-to-source code instrumentation. Our approach works transparent and requires no manual changes to the protected application. The two main components, the pre- and the post-processor, are central entities which are configured by policy files. Therefore, these policies establish a central point to administrate the security properties of the web application.

Using our approach web applications can be effectively protected against code injection attacks without requiring profound changes in the application's source code or existing infrastructure.

5. REFERENCES

[1] Maksymilian Arciemowicz. Bypass xss filter in phpnuke 7.9. mailing list BugTraq, <http://www.securityfocus.com/archive/1/419496/30/0/threaded>, December 2005.

[2] Blwood. Multiple xss vulnerabilities in tikiwiki 1.9.x. mailing list BugTraq, <http://www.securityfocus.com/archive/1/435127/30/120/threaded>, May 2006.

[3] 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.

[4] 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.

[5] Ian Jacobs, Arnaud Le Hors, and David Raggett. Html 4.01 specification. W3C recommendation, November 1999.

[6] 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.

[7] Amit Klein. Cross site scripting explained. White Paper, Sanctum Security Group, <http://crypto.stanford.edu/cs155/CSS.pdf>, June 2002.

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

[9] V. 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.

[10] 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*, 2005.

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

[12] Alex Pigrelax. Xss in nested tag in phpbb 2.0.16. mailing list BugTraq, <http://www.securityfocus.com/archive/ 1/404300>, July 2005.

[13] H. G Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[14] Ivan Ristic. *Apache Security*. O'Reilly, March 2005.

[15] RSnake. Xss (cross site scripting) cheat sheet - esp: for filter evasion. Website, <http://ha.ckers.org/xss.html>, last visit 18/08/06.

[16] George Schlossnagle. *Advanced PHP Programming*. Sams, February 2004.

[17] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of 11th ACM International World Wide Web Conference*, pages 396 – 407. ACM Press New York, NY, USA, 2002.

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

[19] 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.