Inlined Monitors for Security Policy Enforcement in Web Applications

Fotios Rafailidis
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
frafaili@yahoo.gr

Ioannis Panagos
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
ioanpana@csd.auth.gr

Alexandros Arvanitidis
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
arvanian@csd.auth.gr

Panagiotis Katsaros
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
katsaros@csd.auth.gr

ABSTRACT

Improper input validation in Web Applications undermines their security and this may have disastrous consequences for the users. Input data can or cannot be harmful depending on how they are used with regard to the interactions with the clients and the accessed sensitive resources (e.g. databases and files). Existing application frameworks cannot guarantee safe input sanitization with respect to all vulnerabilities. Also, when legacy code is incorporated that was not originally written for the Web, its security hardening is costly and error-prone. We propose a reference monitor inlining approach that treats input injection vulnerabilities as a cross-cutting concern. Our monitors enforce high-level security policies for taint propagation control, by weaving checks and repair actions into the untrusted code. Taint policies are specified into JavaMOP, a programming framework for generating runtime monitors, which are weaved into the application through the automated Aspect Oriented Programming process. When monitor design is guided by preliminary static taint analysis, the incurred overhead can be reduced. Further improvements are feasible through JavaMOP's optimizations. As a proof of concept, we present the design and experimental validation of inlined monitors against SQL injection and cross-site scripting attacks.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

http://dx.doi.org/10.1145/2491845.2491861
PCI 2013, September 19 - 21 2013, Thessaloniki, Greece

Keywords

Software Security, Inlined Reference Monitors, Dynamic Taint Analysis, Aspect-Oriented Programming

1. INTRODUCTION

Web Application vulnerabilities such as cross-site scripting (XSS) and SQL injection are caused by flawed coding in combination with improper sanitization of user input and application output. Frameworks and security libraries that are used today in web development aim to automate sanitization, in order to avoid ad hoc security fixes by the developers. However, web frameworks are designed based on assumptions that may be not adequate for all types of SQL injections and XSS attacks. If the application incorporates legacy code that was not originally written for the Web, then ad hoc security hardening by the developer cannot be avoided. This task is error prone, because developers are usually based on manual analysis of how the input and output data are used. Instead of modifying the code, we believe that security should be treated independently from the application logic. Moreover, the defence should not depend on fixed assumptions that are usually made in frameworks and libraries and should be possible to adapt into new threats.

In this article, we introduce an approach for security policy enforcement at runtime. Policy enforcement is based on reference monitor inlining [10] through Aspect-Oriented Programming (AOP). A high-level policy specification in the JavaMOP framework [17] determines where the security code is inserted and what security state is added to the application. Though security is treated as a cross-cutting concern independent from the application logic, the JavaMOP generated aspect code coexists as a part of the application. Our Inlined Reference Monitors (IRMs) can thus mediate the execution of any statement as opposed to alternatives based on external process monitoring, which cannot enforce the whole range of possible security policies through IRMs. Reference monitor inlining also has the potential to be more efficient than external monitoring, since it is free of overhead for context switching to the operating system environment. Moreover, in the JavaMOP framework that was

designed specifically for monitor-oriented programming, we can utilize advanced features for optimizing the monitoring overhead.

Security policies for mitigating XSSs and SQL injections express taint propagation control properties. They are specified by a number of monitored events tracking how string data propagate between program objects from the points where they enter the program as tainted input (sources) to points where they are used for security critical operations (sinks). If the application fails to prevent taint propagation before executing a critical operation, then our IRM calls a sanitization method from a security library. We present IRMs for SQL injection and XSS attacks and provide evaluation results for their effectiveness and efficiency using the Wavsep attack benchmark [12] and real Web applications.

The paper is organized as follows. Related work is reviewed in Section 2. Section 3 discusses the problem of code injection attacks in Web applications. Section 4 explains the monitor-oriented programming approach for security policy enforcement. Section 5 describes our runtime monitoring solution for taint propagation control and Section 6 presents in detail our IRMs for SQL injection and XSS attacks and their evaluation. The paper concludes with a summary of the pros and the weaknesses of our approach and future research directions.

2. RELATED WORK

Related work for countering Web application vulnerabilities propose static or dynamic analysis approaches, as well as solutions that combine both of them.

2.1 Static analysis

`FindBugs` [16] is a widely used tool that provides support for developing bug detector plugins, which can statically analyze Java applications in search of flawed code patterns or erroneous control flows. FindBugs detectors for Web application vulnerabilities do not guarantee the absence of false negatives and they often generate false positives, as it happens with all static analyses [6]. However, the bug detectors for Web application vulnerabilities do not need code annotations, as is the case with many other static analyses. `Lapse+` [24] is another Java code inspection tool for taint propagation flaws without code annotations. A more precise static taint analysis has been recently proposed in [30]. One interesting aspect of the tool implementing this analysis is the provided support for integrating it with Web Application frameworks.

2.2 Dynamic analysis

Dynamic analyses aim to prevent input injection attacks at runtime, thus avoiding the imprecision of static analyses due to their sensitivity characteristics [6]. However, most dynamic analyses do not take into account input sanitization operations that can be either safe or unsafe.

In [15], the authors introduce a dynamic analysis based on positively tainted input. Their tool called `Wasp` prevents possible SQL injection attacks. Data from trusted sources are marked as such and the trust propagates to other data through string manipulation operations. An SQL parser is invoked before executing queries that recognizes SQL keywords, operators and literals. All literals are expected to originate from trusted data, thus resulting in a conservative protection measure. For the developers, it is necessary to define the trusted sources.

In [4], the authors propose a technique to prevent SQL injection attacks by comparing the SQL statement parse tree before and after completing it with the user's input data. This function is provided in the form of a library called `SQLGuard`. It is necessary to rewrite the application's code, in order to introduce library calls as appropriate.

In [26], the authors describe an approach to prevent SQL, XSS and shell injection attacks by modifying the Web applications' API platform. The proposed platform called `CSSE` provides appropriate methods to store metadata for user-provided input, string operations and string evaluations regarding their taint properties. Due to a risk for potential errors, it is recommended the described modification to be applied by experienced security professionals.

In contrast to the discussed dynamic analysis approaches, our IRMs do not require code rewriting, data annotations or modification of the API platform by a security expert. IRMs are integrated into the Web Applications by taking advantage of the automated AOP code weaving process.

2.3 Combined static and dynamic analysis

Approaches that combine static and dynamic analysis aim to build efficient runtime monitors based on information gathered through static analysis. In [13], the authors present a tool called `Amnesia` for mitigating SQL injection attacks. The tool provides support for external process monitoring based on application specific models (regular expressions) that are statically built for the legitimate SQL queries. The applied static analysis yields over-approximations of the acceptable strings in SQL queries. External monitoring is accompanied by comparatively higher runtime overhead than internal monitoring. A limitation of this approach is that safe sanitization of user input is not taken into account for eliminating unnecessary overhead.

In [3], the authors present `Saner`, a tool aiming to prevent SQL and XSS injection attacks in PHP web applications. The proposed approach is based on `Pixy`, a static analysis tool that can detect faulty sanitization methods. At runtime, an external automaton monitor prevents SQL and XSS routines from using unsanitized user input.

In [21], the author introduces `SecuriFly`, a tool based on a Program Query Language (PQL) to express vulnerability specifications. PQL queries are used to generate nondeterministic automata monitors capable to collect information for relevant program events. The number of dynamic checks and the associated overhead is reduced by an appropriate static analysis. Upon a detected attack, it is possible to execute a user-defined action. Monitoring efficiency is highly dependent on the static analysis, which has been criticized that it cannot be easily applied in all circumstances.

The IRMs proposed in this article inherit a series of optimization options provided by the JavaMOP framework, while a preliminary static analysis can be also used for further efficiency improvements.

3. CODE INJECTION ATTACKS

Web applications are executed over a client-server architecture with the web browser in the client role for rendering the user's interface and the web server in the server role. Client-server interactions take place on top of the HTTP protocol, with the http requests triggering the computation of http responses on the server side.

Code injections refer to a class of attacks that exploit user interactions in order to inject malicious code and trap the application into executing it. Such an attempt may succeed if the application lacks appropriate validation of input and output data. In this case, the attacker may be able to append harmful data (strings, characters, etc.) into a cookie, an http request or a response. Some well-known types of code injection attacks are:

- the SQL code injection
- the LDAP injection
- the SSI injection
- the XSS attacks
- the command injection attacks

Input entry program locations such as the receipt of http requests and all cookie read operations are the sources of data that are propagated throughout the program's control flow to locations called sinks, where some derived data is consumed. An attacker can inject malicious data with various techniques [21], like for example parameter tampering, url manipulation, hidden field manipulation, http header tampering or cookie poisoning. The data reaching some sink is used to access the application's state (e.g. a database) or to affect its output (e.g. the dispatch of http responses). If this data is infected by a malicious input that exploits the application's behaviour, this can have harmful consequences for the users.

More specifically, SQL injection attacks succeed, when some specially crafted input is used by the application in dynamically constructed SQL queries that are executed in a back-end database. In this case, the consequences vary from revealing the database structure to reading or modifying sensitive data or even executing database administration commands. The code in Listing 1 reads from an http request the *firstName* of the user and appends it into a SELECT query. If an attacker enters the string `' OR '1'='1`, this traps the application into executing a query that contains the tautology `firstName = ' ' OR '1'='1'` in the `WHERE` clause. In this way, the attacker accesses sensitive information stored in the database.

Listing 1: Sample code for SQL Injection

```
String name = req.getParameter("firstName");
Connection conn = ...
String query = SELECT * FROM authors
       WHERE firstName = '+name+';
Statement stmt = conn.createStatement();
stmt.executeQuery(query);
```

In XSS attacks, an attacker inserts malicious scripts in dynamically generated web pages. When such a script is executed on a user's web browser, it may change the content of the html page, steal cookies or session tokens and may also access or even modify sensitive information such as user account credentials. XSS attacks are categorized into stored, reflected or DOM based attacks, with each one having different consequences. The code in Listing 2 [23] provides a script, which can be used in an input included in a dynamically generated page to steal a cookie from an authenticated user. This is a typical example of a reflected XSS attack.

Listing 2: Sample script for XSS

```
<SCRIPT type="text/javascript">
var adr = '../evil.php?cakemonster=' +
                  escape(document.cookie);
</SCRIPT>
```

4. INLINED MONITORS FOR SECURITY POLICY ENFORCEMENT

4.1 Inlined Reference Monitors

A security policy is a set of constraints on application functions, as well as constraints on access by external entities and access to sensitive data, in order to protect the application against malicious attacks. Security policies are enforced by appropriate mechanisms that observe the program execution and take remedial actions if a policy violation is detected. These mechanisms are called reference monitors and they are designed to fulfill the following criteria:

- All actions relevant to the policy are recorded.
- Internal or external actions that may threaten the monitor's integrity should be excluded.
- It should be possible to analyze and prove the monitor's correctness.

Reference monitors keep track of the necessary security state information for policy enforcement, which can be only modified through a well-defined set of security updates. Various validity checks take place for the policy's security properties, in order to prevent illegal actions that attempt to violate any of them. Every single update is enacted by a specific monitored event. This association with the observable events in the application's execution history is realized through a high-level policy specification.

`IRMs` integrate the security control mechanism with the application functionality by embedding it into the program's code (source code or executable) [10]. Program locations where IRMs are attached are called insertion points. The code rewriting process is automated through Aspect Oriented Programming. If some monitored events are composed from some other simpler events, these relationships affect the code rewriting process for security policy enforcement.

4.2 Aspect-Oriented Programming

`AOP` [20] offers programming methods and tools that support modularization of program cross-cutting concerns at the level of the source code. AOP implementations provide appropriate expressions that encapsulate each concern in separated *aspect* code.

AspectJ [19] is an AOP extension for Java that allows to specify and attach aspect code into a set of control flow points, known as *join points*. A *pointcut* is a collection of join points that refers to a function and its associated parameter and return values. The code snippet attached to a given pointcut is called *advice* and it is executed when a join point of the pointcut is reached. An aspect consists of the pointcut and its corresponding advice code.

A pointcut may be declared as one out of two possible types, namely *call* or *execution* pointcuts. JavaMOP generated aspects are call-based, i.e. the advice code is executed

upon method calls. Two primitives are used in pointcut definitions. With the `args` primitive the method's parameter values are defined and with the `target` primitive we specify the object for which the pointcut's method is called.

Advice code may be executed with one of the following ways: (i) `before` program's join points, (ii) `after` program's join points regardless of whether the called methods do or do not return normally, (iii) `after returning`, i.e. advice code executed only if a join point method returns normally, (iv) `after throwing`, i.e. executed upon a join point method exception and (v) `around` program's join points if we want to modify the execution context of the called methods. Detailed information on using AspectJ can be found in [31].

The steps towards implementing IRMs based on AOP are:

1. Define the pointcuts needed for security policy enforcement. This definition determines the program's insertion points.

2. Provide advice code that keeps track of the security state and implements appropriate validity checks and repair actions.

3. Use the aspect compiler to identify the program's insertion points, where the IRMs' advice code is automatically weaved.

4.3 Monitor-Oriented Programming

Monitor-oriented programming (MOP) [7] is a software development and analysis framework that supports runtime monitoring. Monitors are automatically synthesized from high-level event specifications and formally expressed properties over the events. The generated monitors are then integrated into a program to check its dynamic behaviours during execution. When a monitored property is validated at runtime, appropriate user-defined actions are triggered and upon a property violation, a repair may be executed.

`JavaMOP` [17, 1, 18] is an instance of the MOP framework that supports runtime monitoring for Java programs. JavaMOP specifications are translated into AspectJ code for monitoring. This code is weaved into the target program by an AspectJ compiler. The whole programming process is shown in Figure 1.

JavaMOP's high-level specifications hide the implementation details for coordinating the execution of program events from the developer. A direct implication is that steps 1 and 2 of the AOP process for IRMs (Section 4.2) are significantly simplified. Moreover, it is easier to implement advanced monitoring policies for improved efficiency such as those that are already provided as JavaMOP options. Monitored security properties can be expressed in one of the supported formal languages including Finite State Machines, Context-Free Grammars, Linear Temporal Logic and others.

5. DYNAMIC TAINT ANALYSIS

Dynamic taint analysis [28] tracks the information flow between sources and sinks at runtime. Data from selected program sources are considered as tainted. Program data that depend on tainted values are marked as such and the rest of them are considered as untainted. Tainted data are not allowed to infect sensitive program sinks. Advanced *taint policies* may enforce repair actions such as the sanitization of tainted data upon reaching a sensitive sink.

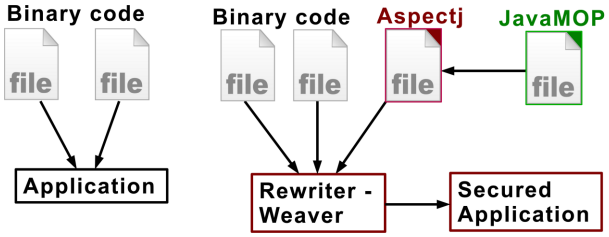

Figure 1: JavaMOP and AspectJ code weaving

A taint policy specification involves the monitoring of three disjoint sets of program events: (a) data entry through tainted sources, (b) operations that propagate the taint to values derived from tainted data or else stop it and (c) operations consuming data at some program sink. Events in each set depend on the language and the type of the monitored program (e.g. locally executed or server-based, user role etc.), as well as on the mitigated security problem (e.g code injection, unpacking malware etc.).

In web applications, data entry events are distinguished in those that concern user input and those corresponding to read operations from trusted sources (e.g. secured cookies, protected files etc.). A clear demarcation between trusted and untrusted sources has to be defined for a given type of monitored program and a targeted security vulnerability.

Events that propagate or stop the taint are distinguished in those that transfer the taint label between program data and those that remove it, as a consequence of a data validation or sanitization operation. The definition of taint removal events is crucial for the correctness of a taint policy. Data validation or sanitization operations either can take or fail taking into account the context, in which tainted data will be used. This may cause missing an existing information flow to a sink (undertainting) or applying an improper sanitization that violates the integrity of some valid data. A safe set of taint removal events for a general taint policy has to be restricted only to standard validation and sanitization operations provided by the application's API. On the other hand, if a taint analysis only adds taint and ignores taint removal events, this causes the problem of taint spread: more and more data become tainted as the program executes with steadily diminishing precision (overtainting).

Data consumed at some program sink can reach it directly from an untrusted program source or can have been generated from at least one preceding propagation event. The policy can simply prevent using tainted data in security sensitive sinks or alternatively it can apply a sophisticated repair operation, if available (e.g. safe data sanitization).

Dynamic taint analysis aims to determine exact taint values over concrete program execution paths, as opposed to static taint analysis that either over- or under-approximates taint over all possible execution paths [22]. Having described the precision challenges for dynamic taint analysis, it is also important to stress that high precision is usually accompanied by very high monitoring overhead especially for compute-bound applications [27]. Overhead reduction can be guided by preliminary static taint analysis, in order to insert IRMs only for the vulnerable execution paths [5].

6. IRMs FOR SQL INJECTION AND XSS

IRMs generated with the JavaMOP framework have been

previously presented in [17] for a series of application security policies. A draft taint policy specification for SQL injection attacks is proposed, not covering all necessary data entry, taint propagation/removal and data consuming events for a practically applicable and effective solution. The taint policies described in this section complete the aforementioned specification and apply appropriate repair operations for two different Web application attacks, namely SQL injection and XSS. Effectiveness of the generated IRMs is evaluated over a benchmark with many different attack cases and their monitoring efficiency is measured over real-world Web applications.

6.1 Program events for dynamic taint analysis

The *input* event set of the JavaMOP specifications for the SQL injection and XSS IRMs includes the following pointcut methods, used to retrieve parameter values from http requests:

- `HttpServletRequest.getParameter()`
- `HttpServletRequest.getParameterValues()`
- `HttpServletRequest.getAttribute()`
- `HttpServletRequest.getParameterMap()`

Parameters that deliver user input, which is consumed in SQL queries or in dynamically constructed html pages, are considered as untrusted and their values are marked as tainted. This is implemented in appropriate after returning advice code. Demarcation of trusted sources from untrusted ones is based on a preliminary static taint analysis. To this end, we have used the Lapse+ tool.

The *taint removal* event set of the JavaMOP specifications for the IRMs includes the following pointcut methods:

- `Integer.Integer()`, `Integer.parseInt()`
- `Float.Float()`, `Float.parseFloat()`
- `Double.Double()`, `Double.parseDouble()`
- `Short.Short()`, `Short.parseShort()`
- `Long.Long()`, `Long.parseLong()`
- `Date.parse()`, `DateFormat.parse()`, `SimpleDateFormat.parse()`
- `BigInteger.BigInteger()`
- `BigDecimal.BigDecimal()`

The aforementioned Java API methods provide a standardized way to validate their parameter with respect to its assumed data type. For tainted data that are found valid, the executed after returning advice code removes the attached taint label. The JavaMOP specification can be easily extended with more taint removal events, but such an attempt will have to consider the possible effects on the taint analysis safety, as discussed in Section 5.

Taint propagation takes place upon creation of new strings from existing tainted ones. The *propagate* event set for the IRMs includes the following pointcut methods:

- `StringBuilder.StringBuilder()`, `StringBuilder.append()`
- `StringBuffer.StringBuffer()`, `StringBuffer.append()`
- `String.String()`, `String.concat()`

In advice code executed before these methods, the taint label(s) of the parameter(s) are checked and an appropriate label is associated with the returned string.

The *usage* event set for the SQL injection IRM consists of the following pointcut methods used to access security critical database resources:

- `Statement.executeQuery()`
- `Statement.executeUpdate()`
- `Statement.addBatch()`
- `Statement.execute()`

Finally, the corresponding event set for the XSS IRM includes methods used to append text in dynamically generated html pages:

- `JspWriter.println()`, `JspWriter.print()`
- `PrintWriter.println()`, `PrintWriter.print()`, `PrintWriter.write()`, `PrintWriter.append()`
- `BufferedWriter.write()`
- `CharArrayWriter.write()`, `CharArrayWriter.append()`
- `OutputStreamWriter.write()`
- `PipedWriter.write()`
- `StringWriter.write()`, `StringWriter.write()`
- `FilterWriter`

For both event sets, if the method parameters contain tainted data, then our IRMs either prevent a potential attack or they enforce an appropriate repair operation. This is implemented in advice code executed around the program's join points.

6.2 JavaMOP specifications

The IRMs security state is encoded in two maps, defined as shown in Table 1. The `TaintedInput` map stores all data originated from untrusted sources. For efficiency reasons, monitored data include only those retrieved from the parameters in `ParamName` of the input event methods. As we already mentioned, these parameters can affect security critical operations and they are identified by a preliminary static analysis. The `TaintedStrings` map stores only the tainted strings derived from propagation events.

The pseudocode of Algorithm 1 summarizes the IRM advice code executed for every method in the input event set. Upon a method call, if the parameter name is a member of `ParamName`, the return value stored in `TaintedInput` is associated with an instance of `StringData` with `isTainted` set to `true`.

Algorithm 2 shows the monitoring logic for every method in the taint removal event set. If a method call returns a value of some valid data type and the used parameter

Table 1: IRM security state information

`StringData <isTainted, taintedData>`	`isTainted`: boolean `taintedData`: Set of tainted data that infect a string
`ParamName<String>`	Set of monitored parameter names
`TaintedInput <String, StringData>`	Map for the untrusted input data
`TaintedStrings <String, StringData>`	Map for tainted strings from propagation events

Algorithm 1: *After returning* advice for the *input* event

```
1 Input: tainted ;                    // Returned from method
2        param ;                      // Parameter of method
3 sd ← new StringData;
4 if param in ParamName <String> then
5 |   sd.isTainted ← true;
6 |   put <tainted, sd > in TaintedInput;
```

is a member of `TaintedInput`, then `isTainted` changes to `false`.

Algorithm 3 summarizes the IRM advice code executed for every method in the propagate event. The code comprises three conditional branches. The parameter value can be (i) a member of `TaintedInput`, (ii) a member of `TaintedStrings` or (iii) is not included in any of these and the method is invoked for a member of `TaintedStrings`. `TaintedStrings` is updated as appropriate for each of the mentioned cases.

Algorithm 4 shows the monitoring logic for every method in the usage event set. For a method parameter in `TaintedStrings`, its associated `StringData` instance is retrieved. If `isTainted` is `true`, then all members of `taintedData` are replaced in the parameter with a sanitized value generated by an `ESAPI` [25, 11] method call. `ESAPI` security libraries provide a series of possible repair actions available for many different programming languages/frameworks, such as PHP, Java, .NET, ASP, ColdFusion, C/C++, Ruby and Perl. For the SQL injection IRM, the `encodeForSQL` filtering method is called. Respectively, sanitization for JavaScript Web applications against XSS takes place through an `encodeForJavaScript` method call. Specialized html sanitization checks can be applied by the `encodeForHTML` and `encodeForHTMLAttribute` methods.

Two JavaMOP options are utilized in order to improve monitoring efficiency as much as possible:

- `creation` option
  Monitoring starts upon the occurrence of a creation event and ignores previous events. In our IRMs the

Algorithm 2: *After returning* advice for the *taint removal* event

```
1 Input: param ;                      // Parameter of method
2        rValue ;                     // Returned from method
3        sd ;                         // Instance of StringData
4 if rValue not null then
5 |   if <param, sd > in TaintedInput then
6 |   |   sd.isTainted ← false;
```

Algorithm 3: *Before* advice for the *propagate* event

```
1  Input: param ;                     // Parameter of method
2         target ;                    // Invoked object
3         propagatedData ;            // Propagated from method
4         sd, sd1, sd2, sd3 ;         // Instances of StringData
5  if <param, sd1> in TaintedInput then
6  |   if sd1.isTainted then
7  |   |   if <target, sd2> in TaintedStrings then
8  |   |   |   sd2.isTainted ← sd1.isTainted;
9  |   |   |   add param in sd2.taintedData;
10 |   |   |   put <propagatedData, sd2> in TaintedStrings;
11 |   |   else
12 |   |   |   create sd of StringData;
13 |   |   |   sd.isTainted ← sd1.isTainted;
14 |   |   |   add param in sd.taintedData;
15 |   |   |   put <propagatedData, sd > in TaintedStrings;
16 |   else if <target, sd3> in TaintedStrings then
17 |   |   put <propagatedData, sd3> in TaintedStrings;
18 if <param, sd1> in TaintedStrings then
19 |   if sd1.isTainted then
20 |   |   if <target, sd2> in TaintedStrings then
21 |   |   |   sd2.isTainted ← sd1.isTainted;
22 |   |   |   add every value of sd1.taintedData in
   |   |   |   sd2.taintedData;
23 |   |   |   put <propagatedData, sd2> in TaintedStrings;
24 |   |   else
25 |   |   |   put <propagatedData, sd1> in TaintedStrings;
26 |   else if <target, sd3> in TaintedStrings then
27 |   |   put <propagatedData, sd3> in TaintedStrings;
28 if <target, sd3> in TaintedStrings then
29 |   put <propagatedData, sd3> in TaintedStrings;
```

Algorithm 4: *Around* advice for the *usage* event

```
1 Input: param ;                      // Parameter of the method
2        sd ;                         // Instance of StringData
3 if <param, sd > in TaintedStrings then
4 |   if sd is tainted then
5 |   |   foreach s_i in sd.taintedData do
6 |   |   |   replace s_i with ESAPI(s_i) in param
```

Table 2: Wavsep injections not properly sanitized

ESAPI methods	GET & POST cases
encodeForSQL	Nr. 6, 14, 15, 16, 17, 18, 19 • err. 500 responses • err.200 responses • err. 200 with differentiation Nr. 1, 2, 3, 4, 5, 6, 7, 8 • identical 200 responses
encodeForJavaScript	Nr. 12, 28

input event is annotated with this option.

- `decentralized` option
  Monitors indexing trees are piggybacked into object states in order to reduce the monitor lookup overhead [7].

AspectJ code weaving is performed only for the class files that include one or more events taking place in some taint infection path. These files were identified through the preliminary static analysis with the Lapse+ tool.

The two IRMs can be easily extended with new parameter names in the set `ParamName`, additional monitored Java methods in the JavaMOP events and different data validation and sanitization methods. If the set `ParamName` is not used, then we obtain an application independent security solution at the cost of a higher policy enforcement overhead.

6.3 Experimental evaluation

Experimental evaluation of the described IRMs was based on two benchmarks. The Wavsep benchmark [12] is a test suite with a series of SQL injection and XSS attacks of many different types. *Our IRMs detected all the attacks* and reported the tainted data reaching the security critical operation.

The used `ESAPI` method `encodeForSql` prevents a type of SQL injection that exploits special characters like "'". Additional measures may be used to validate strings, such as white or black character lists or casting to SQL data types (integer, date etc.). These alternatives can be easily integrated in the IRM based on the parameters that mark an SQL query as tainted and their types. For the XSS IRM, the `ESAPI` method `encodeForJavaScript` prevents also code injections that exploit special characters. Table 2 shows the Wavsep attack cases for which the used `ESAPI` methods do not provide proper sanitization. The `encodeForSql` method failed to repair the SQL attack in 58 out of 130 cases, whereas the `encodeForJavaScript` failed in only 4 out of the 64 attacks.

The second benchmark [14] has been used in the evaluation of the Amnesia toolset for mitigating SQL injection attacks [13]. It consists of five commercial Java Web applications from 5.5 to 16.5 kLOC (Employee Directory, Bookstore, Events, Classifieds, Portal) and two smaller applications with about 5 kLOC (Checkers and OfficeTalk) that were developed by students. *All SQL injection attacks were detected* and the runtime overhead of our monitoring solution was measured. Measurements took place in two different settings:

1. The pure monitoring overhead without taking into account the cost associated with data validation and data sanitization. This metric quantifies only the cost of applying the IRM in order to analyze taint propagation from sources to sinks. The maximum pure monitoring overhead was only 7%.

2. The policy enforcement overhead that accounts also the cost of using the `ESAPI` library for sanitizing tainted values. We observed an additional 10-15% overhead depending on the number of `encodeForSQL` calls (the method is not called for already validated data). The aggregate cost for security policy enforcement is in the order of 20%, corresponding to a CPU time of 40-50ms for the commercial web applications.

To conclude, the measured pure monitoring overhead is negligible, whereas the policy enforcement overhead for using the `ESAPI` library in a real application is affordable.

7. CONCLUSIONS

We presented an approach for defence against SQL injection and XSS attacks by enforcing security policies at runtime. Policy enforcement takes place by reference monitor inlining. This is performed by weaving a security aspect that is automatically generated from a high-level policy specification in the JavaMOP programming framework. JavaMOP is a monitor-oriented programming framework used in runtime verification. Our inlined reference monitors specify taint propagation policies that check how data propagate between program objects and enforce appropriate measures when needed. Experimental evaluation showed 100% success in stopping the SQL injection and XSS attacks of a test benchmark and five commercial Web applications, at the cost of an affordable overhead.

Future research work will focus on further improving monitoring efficiency by utilizing the JavaMOP parametric monitoring mechanism. An important research direction is the formal specification of taint policy monitors [8, 9], in order to reason for their correctness and compositions irrespective of the Web application logic. The described approach can be also applied for the enforcement of security properties beyond those countered by dynamic taint analysis. The so-called temporal safety or typestate properties [29] concern a wide range of application security problems [2, 17] and can be easily expressed with the existing JavaMOP formal specification languages.

8. ACKNOWLEDGMENTS

This work was performed in the framework of the TRACER project, which is partly funded by the Hellenic General Secretariat of Research and Technology (09ΣΥΝ-72-942).

9. REFERENCES

[1] Javamop description. `http://fsl.cs.uiuc.edu/index.php/Special:JavaMOP3`, 2012.

[2] V. Almaliotis, A. Loizidis, P. Katsaros, P. Louridas, and D. Spinellis. Static program analysis for Java card applets. In G. Grimaud and F.-X. Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2008.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner:

Composing static and dynamic analysis to validate sanitization in Web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 387–401. IEEE Computer Society, 2008.

[4] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, pages 106–113, 2005.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50. ACM, 2008.

[6] G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, 0:96–103, 2011.

[7] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588. ACM, 2007.

[8] M. Dam, G. Le Guernic, and A. Lundblad. TreeDroid: a tree automaton based approach to enforcing data processing policies. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 894–905. ACM, 2012.

[9] D. Distefano, R. Grigore, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *TACAS*, pages 260–276, 2013.

[10] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004.

[11] Google. Enterprise security api. `http://code.google.com/p/owasp-esapi-java/`, 2011.

[12] Google. Wavsep benchmark. `http://code.google.com/p/wavsep/`, 2012.

[13] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering*, pages 174–183, Long Beach, California, USA, November 2005.

[14] W. G. J. Halfond and A. Orso. Amnesia testbed. `http://www-bcf.usc.edu/~halfond/testbed.html`, 2005.

[15] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 175–185. ACM, 2006.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[17] S. Hussein, P. Meredith, and G. Roşu. Security policy monitoring and enforcement with JavaMOP. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 3:1–3:11. ACM, 2012.

[18] D. Jin. *Making Runtime Monitoring of Parametric Properties Practical*. PhD thesis, University of Illinois at Urbana-Champaign, August 2012.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[21] B. Livshits, M. Martin, and M. S. Lam. SecuriFly: Runtime protection and recovery from web application vulnerabilities. Technical report, September 2006.

[22] A. Loizidis, V. Almaliotis, and P. Katsaros. Static program analysis of multi-applet JavaCard applications. In *Software Engineering for Secure Systems: Industrial & Research Perspectives*, pages 286–304. IGI Global, 2011.

[23] OWASP. Cross-site Scripting (xss). `https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29`, 2011.

[24] OWASP. LAPSE project. `https://www.owasp.org/index.php/OWASP_LAPSE_Project`, 2011.

[25] OWASP. Enterprise security API. `https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API`, 2012.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*, pages 124–145. Springer-Verlag, 2006.

[27] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on croarchitecture*, MICRO 39, pages 135–148. IEEE Computer Society, 2006.

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331. IEEE Computer Society, 2010.

[29] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[30] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: accurate and scalable security analysis of web applications. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 210–225. Springer-Verlag, 2013.

[31] C. Xerox and R. C. P. Alto. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/next/progguide/index.html`, 2003.