Fortifying Applications Against Xpath Injection Attacks

Dimitris Mitropoulos
dimitro@aueb.gr

Vassilios Karakoidas
bkarak@aueb.gr

Diomidis Spinellis
dds@aueb.gr
Department of Management Science and Technology
Athens University of Economics and Business

Abstract

Code injection derives from a software vulnerability that allows a malicious user to inject custom code into the server engine. In recent years, there have been a great number of such exploits targeting web applications. In this paper we propose an approach that prevents a specific kind of code injection attacks known as xpath injection in a novel way. To detect an attack, our scheme uses location-specific identifiers to validate the executable xpath code. This identifiers represent all the unique fragments of this code along with their call sites within the application.

1 Introduction

Most developers have been trained to write code that implements the required functionality without considering its many security aspects (Howard & LeBlanc 2003). As a result, most vulnerabilities derive from a relatively small number of common programming errors that lead to security holes (Kuang et al. 2006), (Younan 2003). It is very common, for a programmer, to make false assumptions about user input (Wassermann & Su 2004). Assuming only numeric characters will be entered as input, or that the input will never exceed a certain length are some classic examples of such presumptions.

There is a great variety of malicious attacks that derive from such vulnerabilities. One of the most important and damaging class of attacks that must be taken into account when designing and implementing web applications, are code injection attacks (Barrantes et al. 2003), (Kc et al. 2003).

Our proposed scheme fortifies web applications by countering code injection attacks. Particularly the ones that exploit vulnerabilities that are caused by the processing of compromised executable XPath code.

In more detail, our contribution includes:

XPath Code Identification All the possible XPath code that exists in the application, is identified and associated with unique IDs. Based on this, our mechanism prevents the execution of unwanted XPath code.

Detailed Logging All XPath code execution is monitored and logged. Logs can be reviewed later to identify weak spots in the application.

Transparent Usage Our pattern typically acts as a proxy to an already existing application library that implements the XPath functionality.

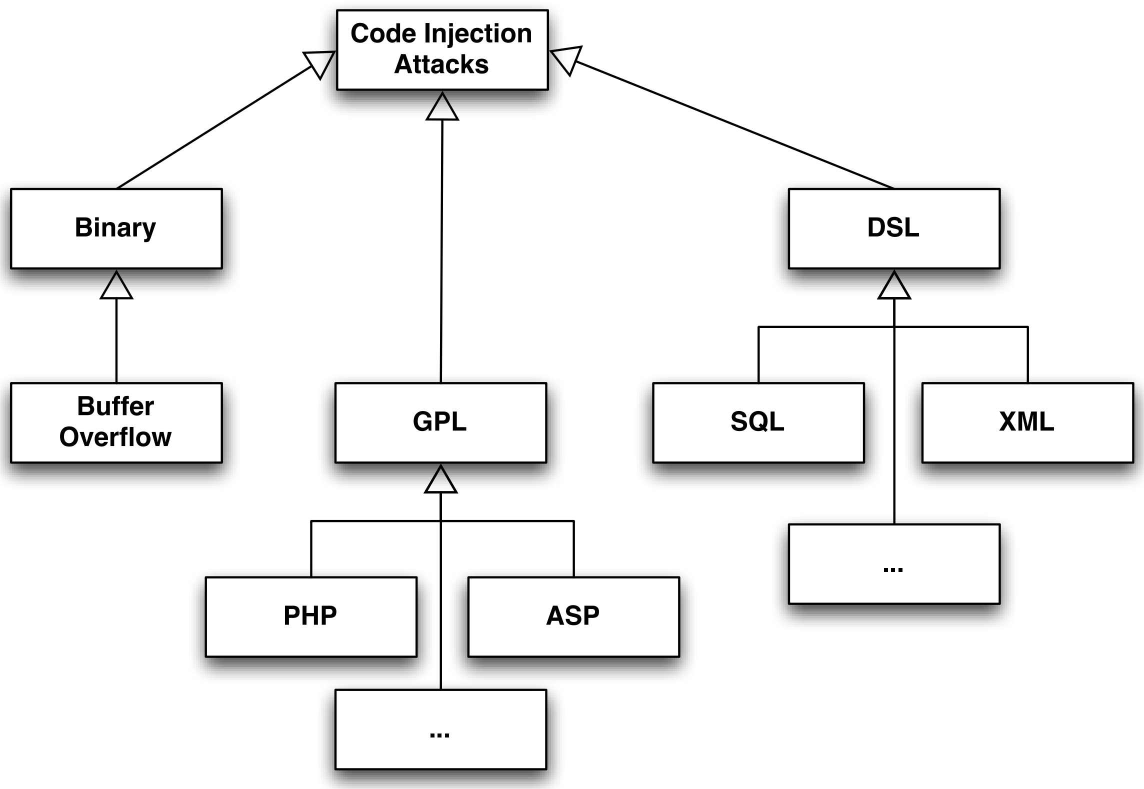

Figure 1: A taxonomy of code injection attacks

The rest of the paper is organized as follows: An introduction to code injection attacks is presented in the next section. In Section 3 we discuss in detail our proposed scheme. Section 4 contains a review of the current related work. In the final section we report the advantages and the disadvantages of our approach and discuss about possible expansions.

2 Code Injection Attacks

Code injection is a technique to introduce code into a computer program or system by taking advantage of the unchecked assumptions the system makes about its inputs (Younan et al. 2005). With this kind of attacks, a malicious user can view sensitive information, destroy or modify protected data, or even crash the entire application (Anley 2002). Such attacks are not restricted to any specific language (Wassermann & Su 2004).

A taxonomy of code injection attacks appears in Figure 1 as a UML class diagram. Binary injection involves the insertion of binary code in a target application to alter its execution flow and execute compiled code. This category, includes the infamous buffer-overflow attacks (Lhee & Chapin 2003).

Another subset of code injection, includes the usage of *General Purpose Languages* (GPL) like PHP and ASP (Hope & Walther 2008). A simple example, could include the control of an input string that is fed into an `eval()` function call. Consider the following fragment written in PHP:

```
$variable = 'foo';
```

```
$input = $_GET['argument'];
// $input = 'delete("/")'; for example
eval('$variable = ' . $input . ';');
```

The argument of `eval()` will be processed as additional commands, since this function is designed to dynamically interpret and execute PHP code.

DSL injection attacks comprise a very critical subset of code injection. Most web applications are typically developed in a General Purpose Language (GPL) along with a Domain Specific Language (DSL) which, is used to address the needs of specific tasks. Hence, languages like SQL and XML play a very important role in the development of every modern web application (Su & Wassermann 2006), (Howard & LeBlanc 2003). For instance, many applications have interfaces where a user can input data to interact with the application's underlying relational database management system RDBMS. This input becomes part of an SQL statement, which is then executed on the RDBMS. A code injection attack that exploits the vulnerabilities of these interfaces is called an "SQL injection attack" (CERT 2002), (Viega & McGraw 2001). One of the most common forms of such an attack involves taking advantage of incorrectly filtered quotation characters. In a login page, besides the user name and password input fields, there is usually a separate field where users can input their e-mail address, in case they forget their password. The statement that is probably executed can have the following form:

```
SELECT * FROM passwords WHERE email = 'emailIgave@example.com';
```

If an attacker, inputs the string *anything' OR 'x'='x*, she could likely view every item in the table.

Another emerging kind of DSL-driven injection attacks are XPath injection attacks (XPIAs) (OWASP 2009). We focus on this category in the following sub-section.

2.1 XPath Injection Attacks

The usage of XML documents instead of relational databases makes web applications vulnerable to XPIAs (Cannings et al. 2007). This is because of the loose typing nature of the XPath language (Benedikt & Koch 2008), (Kimelfeld & Sagiv 2008), (Benedikt et al. 2005). XPIAs can be considerably dangerous since XPath 1.0[1] not only allows one to query all items of the database, but also has no access control to protect it.

Using XPath querying, a malicious user may extract a complete XML document, expose sensitive information, and compromise the integrity of the entire database. Consider the following XML file used by an e-commerce website to store customers' order history:

```
<?xml version="1.0" encoding="UTF-8"?>
<orders>
    <customer id="1">
        <name>Dimitris Mitropoulos</name>
        <email>dimitro@aueb.gr</email>
        <creditcard>12345678</creditcard>
        <order>
        <item>
            <quantity>1</quantity>
            <price>1.000</price>
            <name>television</name>
```

[1] http://www.w3.org/TR/xpath

```
            </item>
        <item>
            <quantity>2</quantity>
            <price>9.00</price>
            <name>CD-R</name>
        </item>
        </order>
    </customer>
...
</orders>
```

The web application allows its users to search for items in their order history based on the price. The query that the application performs could look like this:

```
string query =
"/orders/customer[@id='" + cId + "']" +
"/order/item[price >= '" + pFilter + "']";
```

Without any proper input validation a malicious user will be able to select the entire XML document by entering the following value:

```
'] | /* | /anything[bar='
```

With a simple request, the attacker can steal personal data for every customer that has ever used the application. XPath statements do not throw errors when the search elements are missing from the XML file in the same way that SQL queries do when the search table or columns are missing from the database. Because of the forgiving nature of XPath, it is much more easier for an attacker to use malicious code in order to perform an XPpath injection attack than an SQL injection attack.

3 Design and Implementation

In order to secure an application from XPIAs our mechanism must go through a learning phase. During this phase all the XPath queries of the application must be executed so that we can can identify them in a way we will show in the next section. Then the operation of the mechanism can shift into production mode, where it takes into account all the legal queries and can thereby prevent XPIAs.

3.1 Unique Identifiers

All XPath queries of an application can be identified combining three of its characteristics, namely: its *call stack trace*, its *nodes*, their corresponding *axes* and the various *operators*. The stack trace, involves the stack of all methods from the method of the application where the query is executed down to the target method of our mechanism. The nodes of the statement, is a set that includes various kinds of keywords like: attributes, elements, text etc. The XPpath axes are particular keywords that define a node-set, relative to a specified node. Finally, the operators are the functions that operate on values or other functions.

A formal representation of the application's identifiers that should be accepted as legitimate ones, is the following: If during an application's normal run, K is the set of method stack traces at the point where an XPath statement is executed; N is the set of the nodes; A the set of the corresponding axes, and O the set of the operators, the set of the legitimate query identifiers I is defined as follows:

$$I = \{\omega : \omega = (k, a_1, a_2, ...), k \in K, a_i \in (N \cup A \cup O)\} \quad (1)$$

When the system operates in the training mode, each query identifier Q is added to I. In production mode a query with a identifier Q is considered legal *iff* $Q \in I$. Obviously, a query cannot be singularly identified by using just one of the above characteristics.

To combine these characteristics, when a query is about to be executed our mechanism carries out two actions. First, it strips down the query, removing all numbers and string literals. So if the statement presented in Section 2.1 is being executed the corresponding values for `cId` and `pFilter` will be removed from the query string. The mechanism also traverses down the call stack, saving the details of each method invocation, until it reaches the statement's origins. The association of stack frame data with each XPath statement is an important defense against attacks that try to masquerade as legitimate statements.

3.2 The Proposed Architecture

The proposed design is practically a *proxy* library that contains the default implementation of XPath and has the security features of our approach. Figure 2 depicts our design as a UML class diagram.

Our library acts as a plug-in in JDK's xml implementation. It implements the `abstract` class `XPathFactory` and the interface `XPath`. Specifically, the class `SecureXPathFactory` is the entry point for our library, and the `SecureXPath` is the actual implementation of the XPath application library.

Both classes depend heavily on `javax.xml.xpath` package, that contains the XPath implementation. The class `SecureXPath` implements the following functionality:

Unique Identifiers Each XPath query is stripped down and a *unique identifier* is generated.

Registry The collection of all valid *unique identifiers* for an application. The registry can use various back-ends to store its data, like databases, flat files etc.

The *Registry* is populated with the XPath queries during the *training mode*. Before the deployment of the application, the development and the testing teams should execute the application many times in order for our secure layer to record all the valid queries. These are stored in the registry and are used in the *production mode* for the query validation.

3.3 Injection-detection Scheme

Figure 3 illustrates the proposed injection detection scheme as a UML activity diagram. The *xpath proxy library* accepts the request to execute XPath code from the application. If the training mode is activated, then the XPath code is analyzed and the location-specific identifier is generated. The location-specific identifier is then registered in the XPath query registry as valid. After the training phase the application's flow is followed as designed.

If the *xpath proxy library* is not in training mode, the XPath code is analyzed again and the system checks if the XPath query is valid. Actually, the location-specific identifier is generated again and is compared against all entries in the registry for validity. If not, specific details regarding the invalid call is logged. Upon validation, the XPath code is forwarded to the *xpath library* and is executed. The results are returned transparently to the application.

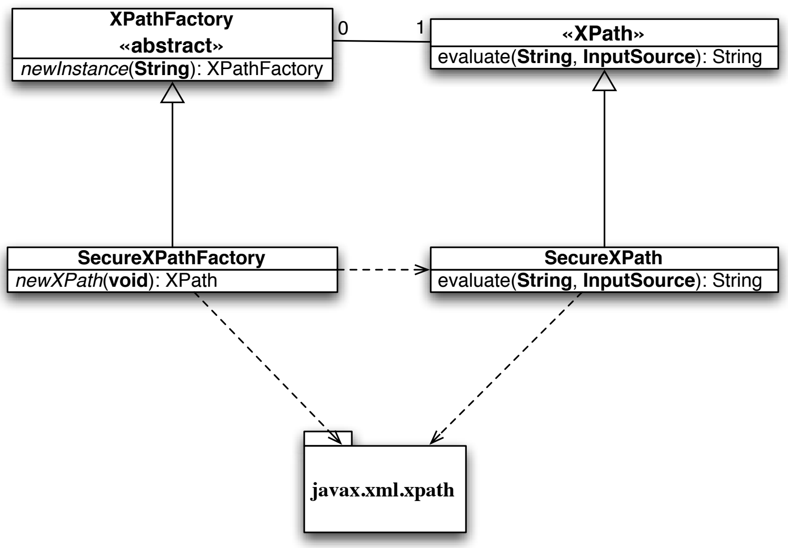
XPathFactory
«abstract»
newInstance(String): XPathFactory
0
1
«XPath»
evaluate(String, InputSource): String
SecureXPathFactory
newXPath(void): XPath
SecureXPath
evaluate(String, InputSource): String
javax.xml.xpath

Figure 2: Overall System Design

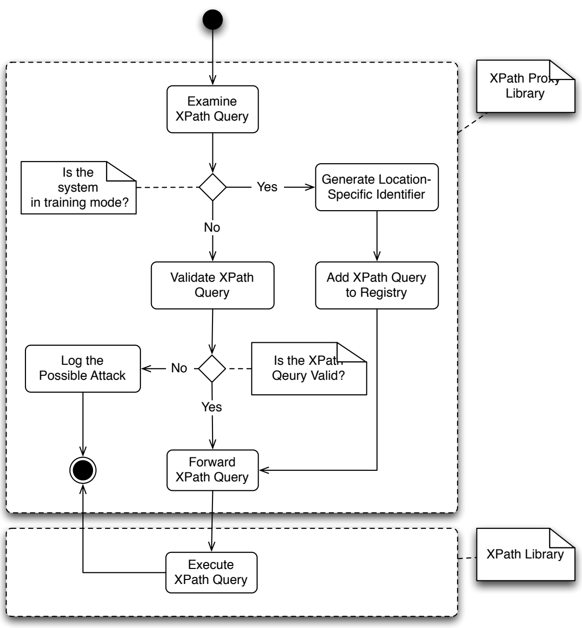

Figure 3: Proposed Prevention Mechanism

3.4 Implementation Details

The *xpath proxy library* depends on `java.xml.xpath`[2] package. A typical usage of XPath library in a Java application is exhibited in the following code fragment.

```
XPathFactory xpf = XPathFactory.newInstance();
XPath xpath = xpf.newXPath();
xpath.compile("//orders/customer");
```

The `XPath` implementation parser is initialised by the appropriate `XPathFactory`. Then the `compile` method is used to instantiate the XPath query. The `XPathFactory` is the implementation of the *Factory* Pattern (Gamma et al. 1994), which permits with an elegant way to support many XPath library implementations.

Our implementation uses this facility to introduce the security library without any significant changes in the source code of the original application. Practically, a developer needs to address two things, in order to fortify his application:

1. Add into his classpath the `jar` file with the secure library.
2. Invoke the appropriate `XPathFactory` class.

For example, the previous code fragment should be modified as follows to include our approach:

```
XPathFactory xpf =
      XPathFactory.newInstance(XPathFactory.DEFAULT_OBJECT_MODEL_URI,
                               "org.sdriver.xpath.SecureXPathFactory",
                               classloader);
XPath xpath = xpf.newXPath();
xpath.compile("//orders/customer");
```

The rest of the code in the application should remain the same.

The *training mode* of the library is activated through the `XPathFactory.setFeature()` method, which is a standard call of the `XPathFactory` abstract class. In our previous example, the *training mode* can be set using the following code:

```
XPathFactory xpf =
      XPathFactory.newInstance(XPathFactory.DEFAULT_OBJECT_MODEL_URI,
                               "org.sdriver.xpath.SecureXPathFactory",
                               classloader);
xpf.setFeature("TrainingMode", true);
XPath xpath = xpf.newXPath();
```

During the *training mode*, the library generates unique identifiers for the queries and add them to the *Registry*, which acts as the database for them and in the prototype implementation is a in-memory `TreeSet` structure. The implementation of the *Regsitry* in the prototype can be easily modified to support any store and retrieval mechanism, according to the application needs.

[2] `http://java.sun.com/j2se/6/docs/api/javax/xml/xpath/package-summary.html`

3.5 Evaluation

Our prototype is build in order to counter many forms of XPIAs including taking advantage of:

1. incorrectly passed parameters
2. incorrect type handling and
3. incorrectly filtered quotation characters

XPath language is usually combined with XSLT applications, thus we were unable to find a simple yet realistic example to test our library in terms of accuracy. In addition, XPIAs are relatively new[3], hence there aren't any real-world web applications that have a record of being vulnerable to this kind of attacks. The prototype implementation[4] of the proposed XPath library introduces functionality that repels code injections, but also adds significant overhead at runtime.

The secure XPath library was tested in laboratory for performance against the standard XPath library shipped with the Java Development Kit (JDK). We have to note that the secure XPath library implementation depends on the JDKs' library. Consequently, the time difference between was clearly introduce by the security layer. It is logical, that the performance of the secure XPath library can be increased with careful optimisation, and the results of our experiments are indicative.

The benchmark was executed on a `Core 2 Duo 2.4Ghz` CPU on a Mac OS X (v10.5). The Java version was 1.6 and the library was compiled with aggressive compile-time optimisations.

The experiment process was simple, we populated the *Registry* with 10 XPath expressions and the invoked the `XPath.compile()` method (standard library call). The iteration number was 1,000,000. The XPath query that we used follows:

```
//EXAMPLE/CUSTOMER[@id='1' and (@type='B' or @type='C')]
```

The benchmark results showed that the library performed 128% slower than the standard XPath library (21,311 msecs against 48,761 msecs). The benchmark was executed five (5) times and the statistical mean was used to produce the above results. In addition, each benchmark iteration had a warm-up period for the virtual machine.

4 Related Work

XPath injection is the unpopular little sibling of SQL injection (Cannings et al. 2007). As a result, the amount of work done for the prevention of SQL injection, is significantly more than XPath (Viega & McGraw 2001), (Halfond et al. 2006). From this countermeasures there is a number of approaches that could be used to counter XPath injection attacks (Cook & Rai 2005), (Lee et al. 2002). Actually, our proposed architecture is based upon our previous work on the prevention of SQL injection attacks (Mitropoulos & Spinellis 2009), (Mitropoulos & Spinellis 2007).

There is also work that intends to cover more than one subsets of code injection covering in an indirect way, the XPath injection issue. One of the first approaches to be introduced was the one that utilizes intrusion-set randomization (Kc et al. 2003). By creating an execution environment that is unique to the running process the authors claim that the application is protected against any kind of an injection attack. A promising scheme that includes the usage of *syntax embeddings* was recently proposed to prevent a

[3]`http://www.ibm.com/developerworks/xml/library/x-xpathinjection.html`

[4]the prototype implementation of the secure XPath library can be found online at `http://gaijin.dmst.aueb.gr/~bkarak/programs/xpath/`

number of DSL-driven injection attacks (Bravenboer et al. 2007). Other approaches include the usage of *software dynamic translation* (Hu et al. 2006), *intrusion block signatures* (Milenković et al. 2005) and the *diversifying* of native APIs (Nguyen et al. 2007).

Due to the wide use of XPath language, which is embedded into several languages for querying and editing XML data, the problem of efficiently answering such queries has received increasing attention from the research community. Apart from the satisfiability of XPath queries (Benedikt et al. 2008), (Groppe & Groppe 2008), (Flesca et al. 2008), academics have made steps to optimize XPath expressions (Balmin et al. 2008), evaluate them (Gottlob et al. 2003) and process them in an efficient way (Gottlob et al. 2005). Still, approaches that can directly counter XPIAs are yet to be proposed.

5 Conclusions and Future Work

In this paper we propose a scheme of preventing XPIAs. The key property of our approach is that every XPath statement can be identified using its location and a stripped-down version of its contents. By analyzing these characteristics during a training phase, we can build a model of the legitimate queries. At runtime our mechanism checks all statements for compliance with the trained model and can block the ones that contain additional maliciously injected elements. The distinct advantage of our approach is that by associating a complete stack trace with the root of each query, can correlate all queries with their call sites. This increases the specificity of the stored query identifiers and avoids false negatives. Another advantage is that our mechanism depends neither on the application nor on the XML data files and can be easily retrofitted to any system. A disadvantage of our approach is that when the application is altered, the new source code structure invalidates existing query identifiers. This necessitates a new training phase.

The similarities of code injection attacks can lead to a more generic way to counter them. Future work on our system involves packaging it in a way that will allow its straightforward deployment, and experimentation along with other approaches that prevent different kinds of injection. It can also involve a further quantitative evaluation of its performance in terms of accuracy and computational efficiency.

References

Anley, C. (2002), *Advanced SQL Injection in SQL Server Applications*, Next Generation Security Software Ltd.

Balmin, A., Özcan, F., Singh, A. & Ting, E. (2008), Grouping and optimization of xpath expressions in db2®purexml, *in* ‘SIGMOD ’08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data’, ACM, New York, NY, USA, pp. 1065–1074.

Barrantes, E., Ackley, D., Forrest, S., Palmer, T., Stefanovic, D. & Zovi, D. (2003), Randomized instruction set emulation to disrupt binary code injection attacks, *in* ‘CCS 2003: Proceedings of the 10th ACM Conference on Computer and Communications Security’, pp. 281–289.
URL: *http://citeseer.ist.psu.edu/barrantes03randomized.html*

Benedikt, M., Fan, W. & Geerts, F. (2008), ‘Xpath satisfiability in the presence of dtds’, *J. ACM* **55**(2), 1–79.

Benedikt, M., Fan, W. & Kuper, G. (2005), ‘Structural properties of xpath fragments’, *Theor. Comput. Sci.* **336**(1), 3–31.

Benedikt, M. & Koch, C. (2008), ‘Xpath leashed’, *ACM Comput. Surv.* **41**(1), 1–54.

Bravenboer, M., Dolstra, E. & Visser, E. (2007), Preventing injection attacks with syntax embeddings, *in* 'GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering', ACM, New York, NY, USA, pp. 3–12.

Cannings, R., Dwivedi, H. & Lackey, Z. (2007), *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions (Hacking Exposed)*, McGraw-Hill Osborne Media.

CERT (2002), 'CERT vulnerability note VU282403', Online `http://www.kb.cert.org/vuls/id/282403`. Accessed, January 7th, 2007.

Cook, W. & Rai, S. (2005), Safe query objects: statically typed objects as remotely executable queries, *in* 'ICSE 2005: 27th International Conference on Software Engineering', pp. 97–106.

Flesca, S., Furfaro, F. & Masciari, E. (2008), 'On the minimization of xpath queries', *J. ACM* **55**(1), 1–46.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Indianapolis, IN 46920.

Gottlob, G., Koch, C. & Pichler, R. (2005), 'Efficient algorithms for processing xpath queries', *ACM Trans. Database Syst.* **30**(2), 444–491.

Gottlob, G., Koch, C., Pichler, R. & Wien, T. U. (2003), Xpath query evaluation: Improving time and space efficiency, *in* 'In Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE03', pp. 379–390.

Groppe, J. & Groppe, S. (2008), 'Filtering unsatisfiable xpath queries', *Data Knowl. Eng.* **64**(1), 134–169.

Halfond, W. G., Viegas, J. & Orso, A. (2006), A classification of SQL-injection attacks and countermeasures, *in* 'Proceedings of the International Symposium on Secure Software Engineering'.

Hope, P. & Walther, B. (2008), *Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast*, O'Reilly Media, Inc.

Howard, M. & LeBlanc, D. (2003), *Writing Secure Code*, second edn, Microsoft Press, Redmond, WA.

Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D., Knight, J. C., Nguyen-Tuong, A. & Rowanhill, J. (2006), Secure and practical defense against code-injection attacks using software dynamic translation, *in* 'VEE '06: Proceedings of the 2nd international conference on Virtual execution environments', ACM, New York, NY, USA, pp. 2–12.

Kc, G. S., Keromytis, A. D. & Prevelakis, V. (2003), Countering code-injection attacks with instruction-set randomization, *in* 'CCS '03: Proceedings of the 10th ACM conference on Computer and communications security', ACM, New York, NY, USA, pp. 272–280.

Kimelfeld, B. & Sagiv, Y. (2008), Revisiting redundancy and minimization in an xpath fragment, *in* 'EDBT '08: Proceedings of the 11th international conference on Extending database technology', ACM, New York, NY, USA, pp. 61–72.

Kuang, C., Miao, Q. & Chen, H. (2006), Analysis of software vulnerability, *in* 'ISP'06: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy', World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, pp. 218–223.

Lee, S. Y., Low, W. L. & Wong, P. Y. (2002), Learning fingerprints for a database intrusion detection system, *in* D. Gollmann, G. Karjoth & M. Waidner, eds, 'ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security', Springer-Verlag, London, UK, pp. 264–280. Lecture Notes In Computer Science 2502.

Lhee, K.-S. & Chapin, S. J. (2003), 'Buffer overflow and format string overflow vulnerabilities', *Softw. Pract. Exper.* **33**(5), 423–460.

Milenković, M., Milenković, A. & Jovanov, E. (2005), 'Using instruction block signatures to counter code injection attacks', *SIGARCH Comput. Archit. News* **33**(1), 108–117.

Mitropoulos, D. & Spinellis, D. (2007), Countering SQL injection attacks with a database driver, *in* T. S. Papatheodorou, D. N. Christodoulakis & N. N. Karanikolas, eds, 'Current Trends in Informatics: 11th Panhellenic Conference on Informatics, PCI 2007', Vol. B, New Technologies Publications, Athens, pp. 105–115.
URL: *http://www.dmst.aueb.gr/dds/pubs/conf/2007-PCI-SQLIA/html/MS07.htm*

Mitropoulos, D. & Spinellis, D. (2009), 'SDriver: Location-specific signatures prevent SQL injection attacks', *Computers and Security* **28**, 121–129.
URL: *http://www.dmst.aueb.gr/dds/pubs/jrnl/2009-CompSec-SQLIA/html/sqlia.html*

Nguyen, L. Q., Demir, T., Rowe, J., Hsu, F. & Levitt, K. (2007), A framework for diversifying windows native apis to tolerate code injection attacks, *in* 'ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security', ACM, New York, NY, USA, pp. 392–394.

OWASP (2009), 'XPATH injection', Online `http://www.owasp.org/index.php/XPATH_Injection`. Accessed, April 30th, 2009.

Su, Z. & Wassermann, G. (2006), The essence of command injection attacks in web applications, *in* 'Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '06', ACM Press, pp. 372–382.

Viega, J. & McGraw, G. (2001), *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, Boston, MA.

Wassermann, G. & Su, Z. (2004), An analysis framework for security in web applications, *in* 'SAVCBS 2004: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems', pp. 70–78.

Younan, Y. (2003), An overview of common programming security vulnerabilities and possible solutions, PhD thesis, Brussel.

Younan, Y., Joosen, W. & Piessens, F. (2005), A methodology for designing countermeasures against current and future code injection attacks, *in* 'IWIA 2005: Proceedings of the Third IEEE International Information Assurance Workshop', IEEE, IEEE Press, College Park, Maryland, U.S.A.
URL: *http://citeseer.ist.psu.edu/article/younan05methodology.html*