Security-aware selection of Web Services for Reliable Composition

Shahedeh A. Khani, Cristina Gacek, Peter Popov

Security-aware selection of Web Services for Reliable Composition

Shahedeh A.khani
Centre for Software Reliability,
City University,
London,
United Kingdom
abdolhossein.shahedeh.1@city.ac.uk

Cristina Gacek
Centre for Software Reliability,
City University,
London,
United Kingdom
cristina.gacek.1@city.ac.uk

Peter Popov
Centre for Software Reliability,
City University,
London,
United Kingdom
p.t.popov@city.ac.uk

***Abstract*— Dependability is an important characteristic that a trustworthy computer system should have. It is a measure of Availability, Reliability, Maintainability, Safety and Security. The focus of our research is on security of web services. Web services enable the composition of independent services with complementary functionalities to produce value-added services, which allows organizations to implement their core business only and outsource other service components over the Internet, either pre-selected or on-the-fly. The selected third party web services may have security vulnerabilities. Vulnerable web services are of limited practical use. We propose to use an intrusion-tolerant composite web service for each functionality that should be fulfilled by a third party web service. The third party services employed in this approach should be selected based on their security vulnerabilities in addition to their performance. The security vulnerabilities of the third party services are assessed using a penetration testing tool. In this paper we present our preliminary research work.**

Keywords*— *Web Services; Selection; Security; Penetration Testing

I. INTRODUCTION

Web Services (WSs) are used to implement Service Oriented Architectures (SOAs). Each service consists of an implementation that is on a network-accessible platform and an interface. The communication with WSs is supported by Simple Object Access Protocol (SOAP) [1], which is a standard protocol for packaging messages before transmitting them through standard Internet technologies. Web Service Description Language (WSDL) [2] is used to describe the interface of a WS, describing the concrete structure of the SOAP messages (description of the operations and their input and output parameters), the service's protocol binding and network location for the WS's implementation (e.g. the URL). SOAP messages and WSDL documents are based on eXtensible Markup Language (XML), which has simplified the interoperability of the various technologies employed in WS development.

WSs enable the composition of independent services with complementary functionalities to produce value-added services, which allows organizations to implement their core business only and outsource other service components over the Internet, either pre-selected or on-the-fly. However, WSs are open to a large population of users therefore, maintaining their security is an important task. Security attacks on WSs may cause unavailability and/or loss of confidentiality and integrity as well as significant monetary penalties. Now the question is what if the selected third party WSs have security vulnerabilities? Vulnerable WSs are of limited practical use. Therefore, security should also be considered in selection and employment of suitable third party WSs.

Our objective is to improve the dependability of composite WSs in which third party WSs are used. Our focus is on the security of WSs. Web services are at risk of security vulnerabilities related to their specific implementation technologies (e.g. XML) as well as those of their underlying platforms (e.g. operating systems and web-services frameworks) and the WS applications themselves (e.g. being vulnerable to SQL injection attacks). Security vulnerabilities related to WSs' implementation technologies are central to the work described in this paper. We propose to use an intrusion-tolerant composite WS (using fault tolerant techniques: N-version programming, diversity) for each functionality that should be fulfilled by a third party WS. In our approach, penetration testing is used to identify security vulnerabilities of available functionally-equivalent candidate third party WSs. Suitable third party WSs will then be selected based on their security vulnerabilities and their performance according to the client's (being the owner of the system) requirements. The selected services will be invoked using an intrusion-tolerant approach as a countermeasure against the security attacks exploiting the vulnerabilities that are not covered/identified by penetration testing.

This paper makes the following contributions:

- It explains our proposed approach in details.
- It exemplifies our approach using a case study.

The remainder of this paper is organised as follows. Related work is discussed in Section II. Section III briefly introduces WSs' specific security vulnerabilities. Penetration testing and the tool we are using are discussed Section IV. Sections V and VI present our proposed approach and the case study respectively. Section VII draws the conclusions and outlines future work.

II. RELATED WORK

Various intrusion detection and prevention methods and approaches have been proposed to secure WSs. However, in addition to the adoption of the proposed methods and

approaches, WSs should also be able to tolerate attacks and continue to provide an acceptable level of service even after intruders have broken in. Intrusion-tolerant systems can be developed using fault-tolerance concepts and approaches.

Redundancy is believed to be a valid defence against physical faults. Running multiple replicas of the system and switching to the functioning one when a failure occurs, is an example of using redundancy to overcome hardware faults [3]–[5]. Redundancy can also be applied to the code, data, and environment of a software system to overcome its nonphysical faults [6]. Design diversity is a recognised defence against design faults. Littlewood, Popov and Strigini [7] have surveyed the benefits of design diversity. Carzaniga, Gorla and Pezzè [8] describe the redundancy as a system's capability of executing the same functionality in several execution environments or in various ways (e.g. using different execution paths). Littlewood and Strigini [9] have argued the validity of using redundancy and diversity for security.

Majorczyk et *al.* [10] have proposed Intrusion Detection Systems (IDS) based on redundancy and diversification of Components-Off-The-Shelf (COTS) and have applied it to web servers. Valdes et *al*. [11] have proposed an intrusion-tolerant web server architecture based on redundant COTS servers running on diverse operating systems and platforms. Gorbenko et *al*. [12] have proposed a generic intrusion-avoidance architecture to be used for deploying WSs in the cloud. This architecture employs software diversity at various system levels and dynamically reconfigures the cloud deployment environment. All above approaches use redundancy and diversity techniques to detect and tolerate the intrusions. However, none of them addresses the attacks exploiting the vulnerabilities caused by the XML standards, which are independent of the type of operating systems or application implementation. Massimo Ficco and Massimiliano Rak [13] have proposed an intrusion tolerance approach for DoS attacks to WSs. It focuses on the detection of attack symptoms, as well as the diagnosis of intrusion effects in order to take appropriate action only when the attack succeeds. This work is more related to our approach. However, it focuses on a specific group of XML DoS attacks, called Deeply-Nested XML.

III. WEB SERVICES' SPECIFIC SECURITY VULNERABILITIES

As stated previously, the communication between WSs is supported by XML-based protocols. This makes WSs vulnerable to XML attacks (vulnerabilities related to their specific implementation technologies). Examples of recently reported security attacks exploiting such vulnerabilities are the attacks on Amazon EC2 SOAP, Eucalyptus cloud WS interfaces [14], [15], different SAML-based frameworks [16] and ciphertext decryption exploitation [17]. Jensen et al. [18], [19] present a list of top WSs' specific security vulnerabilities (related to the implementation technologies). To identify these security vulnerabilities, they have performed exemplary attacks on widespread WS implementations. According to the study, some of these vulnerabilities are due to implementation weaknesses but majority of them are due to protocol flaws. In this section we briefly introduce a number of these security vulnerabilities.

A. Attack Obfuscation

WS-Security [20] is a very flexible security standard that allows signing and encrypting only parts of the message, which contains sensitive data. A disadvantage of using this standard is that the encrypted content may not be inspected without prior decryption. Such encryption can be used by attackers to conceal malicious code. Therefore, if the encrypted part of the message contains an intended attack (e.g. Denial of Service attack.), it will be very difficult to detect.

B. XML Injection

An XML Injection attacker tries to modify the structure of a XML document (e.g. SOAP message) by adding some contents containing XML tags.

C. SOAPAction Spoofing

A SOAP message package consists of a transport protocol header and an envelope. The SOAP envelope consists of a header and a body. The first child element of the body contains the operation addressed by the SOAP request [18]. If the HTTP transport protocol is used, an additional operation identifier element called SOAPAction can be added to the header [18]. This enables the receiving WS to understand what operation the SOAP body contains, prior to XML parsing [21]. However, it is often used as the only qualifier for the requested operation [22].

A WS will be vulnerable to a SOAPAction spoofing attack if the requested operation is identified solely based on the SOAPAction value or first child element of the SOAP body [22]. A successful SOAPAction Spoofing attack will result in unauthorised execution of operations offered by the WS.

D. Denial of Service (DoS) Attacks

Early steps in processing a request SOAP message include parsing and transforming the contents of the message to be usable by the WS's backend applications. Therefore, the XML parser is an essential part of a WS's application logic. Simple API for XML (SAX) [23] and Document Object Model (DOM) [24] are two typical XML parsers.

DOM parsers read the whole XML stream into the memory then create hierarchical objects for each node (an element, an attribute etc.), referenced by the application logic. An attacker can plot a DoS attack on a DOM-based WS by inputting a large XML file [25]. Such attacks (e.g. Oversize payload and Coercive parsing [19]) affect the availability of the WS by exhausting its resources and preventing legitimate users from accessing the service [26]. DOM parsers can also be subject to other types of attacks such as XML injection [25].

On the other hand, SAX parsers perform XML parsing at the start or end of a node without loading the whole XML stream into memory (they load a maximum of two elements into the memory at a time) [25]. Whenever the parser reaches a node, it triggers an event, and the program's event handler

starts processing the data. SAX-based WSs are vulnerable to XML injection attacks [25]. In XML injection attacks the attacker targets the integrity of the XML stream (e.g. SOAP message) by overwriting static portions of it [25]. For example, the attacker modifies the message by adding contents containing XML tags [27].

DoS attacks are one of the most popular attacks, which can be performed through a variety of techniques. This type of attacks exploit the vulnerabilities in XML-based documents (e.g. SOAP messages) targeting the parsing mechanisms and other resources, affecting the availability of the WS. A large number of these attacks, targeting well-known companies such as VISA and PayPal suggests that they can be a serious threat to today's IT infrastructure [28]. Jensen *et al.* [18], [19] have presented a number of DoS attacks.

E. Hash Collision (HashDoS) Attack

Hash tables can be employed within a SOAP message to store values and their references (e.g. attributes and their corresponding namespace). Ideally each key should represent a unique value. If different keys represent the same value, a collision will happen, which results in resource intensive computation. An attacker can exploit a weak hash function to perform a DoS attack [29], [30].

IV. PENETRATION TESTING

Penetration testing or static code analysis approaches can be employed to assess the security of a WS [31]. Penetration testing is an attempt to break into a system not in order to exploit it, but rather to identify its weaknesses [32]. The resistance of the system against penetration testing is a good indicator of its security [27]. In our approach, security vulnerabilities of a third party service play an important role in its selection. Security vulnerabilities related to WSs' implementation technologies are central to our work. Hence, a penetration testing tool called WS-Attacker [27] is chosen for testing candidate third party WSs to identify their security vulnerabilities. The reasons for WS-Attacker selection are: (1) it enables testing for XML-specific security vulnerabilities explained in Section III (2) it performs the attacks automatically and (3) it is an open source penetration testing tool.

WS-Attacker consists of a framework and plugins architecture. Its framework is based on soapUI [32] and sets up an environment for attacking WSs. In WS-Attacker, the attacks are implemented as plugins. Each plugin is an implementation of a model of an adversary performing one type of attack and allows the user to set various parameters, such as number of parallel attack threads, number of requests per thread, milliseconds between every test-probe requests, and milliseconds between every attack requests. WS-Attacker is extendable and provides a plugin interface enabling new attack plugins to be added to the tool [27]. A number of attack plugins have been developed for WS-Attacker by its developers and other researchers [27], [33], [28]. In implementing these plugins, the developers assume that the tester does not have a direct access to the system under attack, and can only examine its vulnerability to an attack by sending payloads to its server then evaluating its response (or its response time in the case of performing DoS attacks). The result (true or false) of performing an attack indicate whether it has been successful [27].

V. PROPOSED APPROACH

As stated previously, WSs are at risk of security vulnerabilities related to their specific implementation technologies (e.g. XML) as well as those of their underlying platforms (e.g. operating systems and web-services frameworks) and the WSs applications themselves (e.g. being vulnerable to SQL injection attacks). Systems could be developed using services offered by different vendors. To explain our approach, we are assuming that we have control over a system that is under development, which should employ third party WSs. The candidate third party WS(s) may have security vulnerabilities. Their security vulnerabilities may be exploited by messages from our system's client, the clients of other systems that are also employing them or their direct clients. The first scenario will definitely affect our system. The last two scenarios could affect our system as a result of security attacks, such as DoS, which may make the third party service unavailable just for few minutes or completely with the need to reboot.

We propose to use an intrusion-tolerant composite WS (using fault tolerant techniques: N-version programming, diversity) for each functionality that should be fulfilled by a third party WS. Our proposed approach includes the following steps:

Step 1. Collect as many WSs as possible, offered by various vendors providing the same functionality as required by the system.

Step 2. Log the failure rate of each service stated in its Service Level Agreement (SLA). Initially the failure rates stated by the provider of the third party service will be considered but this log will be updated every time the service is invoked.

Step 3. Identify their security vulnerabilities using a penetration testing tool (as shown in Fig. 1).

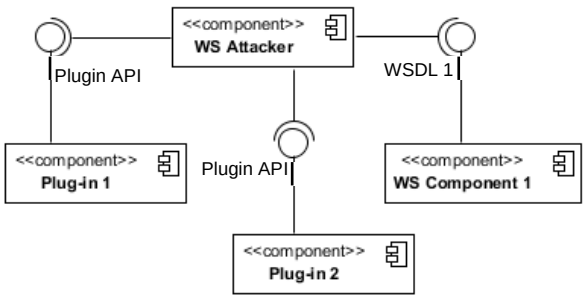

Figure 1: Penetration Testing WSs.

Step 4. Log the vulnerabilities of each service.

Step 5. Create a composite WS out of all available candidate WSs for the functionality that is to be outsourced.

Step 6. For each request received from a client of the system, scan the message to identify which security vulnerabilities it could exploit if it contains malicious contents.

Step 7. Select all third party services without vulnerabilities, which could be exploited if client's request contains malicious contents. The system understands which services should not be selected from the log generated in Step 4. For example a message with SOAPAction elements will exploit the security vulnerability of a third party WS to SOAPAction Spoofing attack if it contains malicious contents. Therefore, no services vulnerable to this type of attack should be selected if the client's request contains SOAPAction elements. The number of the selected WSs at this stage should be greater than or equal to $3f+1$ (f is the number of possible faulty replicas). Castro and Liskov [34], have justified the optimality of $3f+1$ replicas. When the number of WSs without tested security vulnerabilities that could be exploited if client's request contains malicious contents is less than $3f+1$, select the remaining ones from WSs with security vulnerabilities that are of lowest priority to system owner and with lowest failure rates (using the log generated in Step 2).

Step 8. If the number of services selected in Step 7 is greater than $3f+1$, select the services with lowest failure rates (using the log generated in Step 2).

Step 9. Invoke WSs selected in Step 8 concurrently.

Step 10. Re-execute the WSs if all of them fail (Active+Time replication strategy [35]).

Step 11. Terminate the execution of WSs as soon as $2f+1$ responses are returned.

Step 12. Applying majority voting to identify the response that should be returned to the client of the system.

VI. CASE STUDY

To exemplify our approach we are using a stock purchase service. These experiments are run on Intel® Core™ i5-3320M CPU @ 2.60GHz system with 7.88GB usable RAM and 64-bit Operating System. In these experiments only security vulnerability to Coercive parsing attack is considered and the Steps of our proposed approach are taken as follows:

1. We have developed four WSs (two using Apache Axis2 and two using ASP.NET WS frameworks) and selected a third party ASP.NET WS, which provides similar functionality to those we have developed.
2. No information related to failure rate of these WSs was available. Hence, Step 2 of our approach is omitted in this experiment but it will be considered in our future work.
3. Each WS was tested individually for security vulnerability to Coercive Parsing attack by submitting the location of its WSDL file to WS-Attacker then performing the attack with settings (default settings) shown in Table 1.
4. The security vulnerability of these WSs to Coercive Parsing attack was identified (see Table 2). As it is explained in section IV, the developers of WS-Attacker's DoS attack plugins have assumed that the tester does not have direct access to the system under attack, and can only examine its vulnerability to the above DoS attacks by sending payloads to its server then evaluating its response time. They have defined the response time as the time when the last byte of the request is sent to the server until the first byte of the response is received from the server [33]. In designing these attack plugins, all major errors, such as increase in response time caused by variable message sizes or network loads, are eliminated [33]. These attack plugins calculate the median of the response times of the last 10 tampered requests and the median of the last 10 untampered requests. They then work out the ratio of the median response time of the tampered requests to the median response time of the untampered requests. Any ratio notably higher or lower than 1, will be interpreted as a successful attack. Refer to the results presented in Table 2, 100% indicates that the WS has vulnerability to Coercive Parsing attack and 1% shows that it has not this vulnerability.
5. A Business Process Execution Language (BPEL [36]) composite WS (shown in Fig. 2) was created using all all available WSs but only the three ASP.NET services and one of the Axis2 services (four services highlighted in Table 2) were invoked, addressing Steps 5-8 of proposed approach. Steps 6-8 should be performed automatically, which will be addressed in our future work.
6. To address the Steps 9 and 11 of our approach, all four services were invoked concurrently and a variable was dedicated to each of the concurrent processes. Upon receiving a response from each of the services, the corresponding variable would be set to a pre-defined value to indicate that a response is returned. A throw and catch exception handling was employed to terminate WSs execution upon receiving the first three responses (as soon as the dedicated variables to three out of four services are set to the pre-defined value).

The composite service was then tested for security vulnerability to Coercive Parsing attack, using WS-Attacker with the same settings that were used to test individual services. As the results from this experiment illustrate (last column of Table 2), the composite service is no longer vulnerable to Coercive Parsing DoS attack.

The purpose of this experiment was to exemplify our approach as well as testing its effectiveness as a defence against DoS attacks. However, our future work will (a) take into account the failure rate of the WSs (b) scan the client's request (c) performs Step 6-12 automatically (d) employ majority voting.

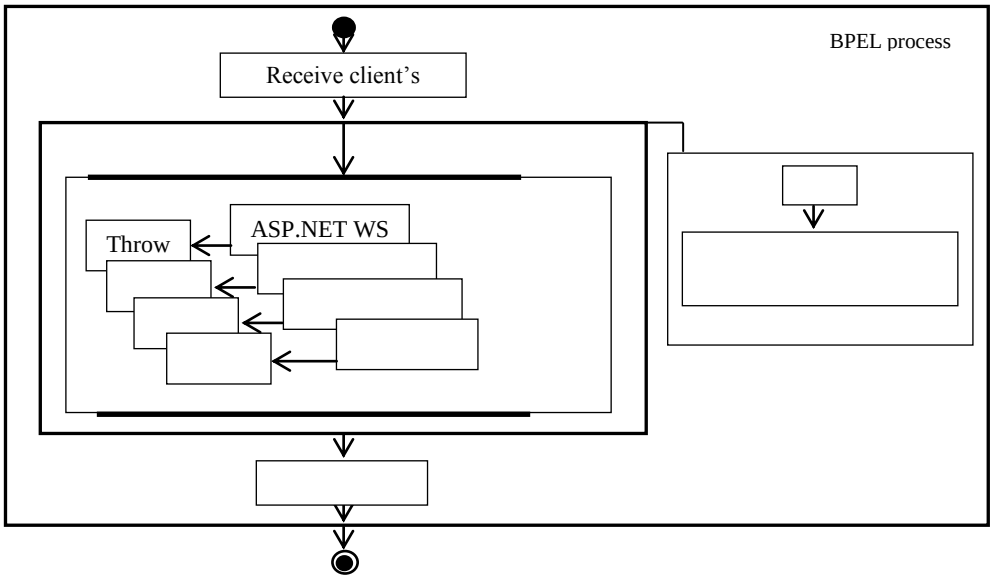

Figure 2: BPEL Diagram of the Composite Service.

Table 1: Coerceive Parting Attack Settings Used for Experiments

Security Vulnerability Type	Common WS-Attacker settings	Specific WS-Attacker settings
Coercive Parsing	2 parallel attack threads, 4 requests per thread, 500 milliseconds between every testprobe request, 750 milliseconds between every attack request, 4 seconds server recovery time, 5 seconds stop after the last tampered request.	75,000 nested elements

Table 2: Preliminary Experimental Results

Security Vulnerability Type \ Web Service Framework	Axis2 web service	Axis2 web service	ASP.Net web service	ASP.Net web service	ASP.Net web service	⇒	Composite service
Coercive Parsing	100%	100%	1%	1%	1%	⇒	1%

VII. CONCLUSIONS

In this paper we proposed to use an intrusion-tolerant composite WS (using fault-tolerant techniques: N-version programming, diversity) for each functionality that should be fulfilled by a third party WS to improve the security of WSs. In our approach, penetration testing is used to identify security vulnerabilities of available functionally-equivalent candidate third party WSs. Suitable third party WSs will then be selected based on their security vulnerabilities and their performance according to the client's (being the owner of the system) requirements. The selected services will be invoked using an intrusion-tolerant approach as a countermeasure against the security attacks exploiting the vulnerabilities that are not covered/identified by penetration testing. We have presented our preliminary experimental results indicating that an intrusion-tolerant composite service may reduce the security vulnerabilities of WS.

Despite using an intrusion-tolerant composite service, we depend on a single orchestration engine, in these experiments BPEL, to manipulate them. Regardless of which orchestration engine we choose, it will always become a possible single point of failure. Thus, if the orchestration engine is vulnerable itself, then it may compromise any composite service that it handles. We note that in our experiments we did not observe this phenomenon. Standard solutions to address this concern would require applying-fault tolerance to the orchestration engines themselves, a problem which we will address in our future work.

We recognise that using the proposed approach introduces an additional delay in processing the requests sent to the BPEL orchestration in comparison to the clients sending the request directly to the component WS. Indeed, with the BPEL orchestration, the client request travels first to the BPEL engine and then it is forwarded to each of the component WSs. The resulting delay may increase significantly and will depend on the quality and speed of the connections.

The use of a composite service based on the best functionally equivalent set of diverse WSs is certainly beneficial for many security concerns (e.g., unavailability given DoS attacks). Nonetheless, if one or more of the underlying WSs has succumbed to an intruder, giving them access to incoming requests, the use of the composite service may actually make us more vulnerable to loss of

confidentiality, as the requests will be sent to all of the underlying services, and not just one of them. Future work should explore how to reduce and/or mitigate this risk. In particular the adjudicator (performing majority voting) will play an important role here. We intend to study its impact systematically.

Currently we are investigating the effectiveness of our approach using WSs developed based on various web services frameworks. In the near future, we will focus on the implementation of a complete composition framework supporting security-aware service selection, composition and adaptation.

REFERENCES

[1] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2," *W3C Recomm.*, vol. 24, 2003.

[2] R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana, "Web services description language (WSDL) version 2.0 part 1: Core language," *W3C Work. Draft*, vol. 26, 2004.

[3] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, Apr. 1962.

[4] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1988, pp. 109–116.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[6] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2005, pp. 235–248.

[7] B. Littlewood, P. Popov, and L. Strigini, "Modeling Software Design Diversity: A Review," *ACM Comput Surv*, vol. 33, no. 2, pp. 177–208, Jun. 2001.

[8] A. Carzaniga, A. Gorla, and M. Pezzè, "Handling Software Faults with Redundancy," in *Architecting Dependable Systems VI*, R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. ter Beek, Eds. Springer Berlin Heidelberg, 2009, pp. 148–171.

[9] B. Littlewood and L. Strigini, "Redundancy and Diversity in Security," in *Computer Security – ESORICS 2004*, P. Samarati, P. Ryan, D. Gollmann, and R. Molva, Eds. Springer Berlin Heidelberg, 2004, pp. 423–438.

[10] E. Totel, F. Majorczyk, and L. Mé, "COTS Diversity Based Intrusion Detection and Application to Web Servers," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, Berlin, Heidelberg, 2006, pp. 43–62.

[11] A. V. Magnus, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stavridou, and T. E. Uribe, "An Adaptive Intrusion-Tolerant Server Architecture," System Design Laboratory, SRI International, CA, 2001.

[12] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky, "Using Diversity in Cloud-Based Deployment Environment to Avoid Intrusions," in *Software Engineering for Resilient Systems*, E. A. Troubitsyna, Ed. Springer Berlin Heidelberg, 2011, pp. 145–155.

[13] M. Ficco and M. Rak, "Intrusion Tolerant Approach for Denial of Service Attacks to Web Services," in *2011 First International Conference on Data Compression, Communications and Processing (CCP)*, 2011, pp. 285–292.

[14] N. Gruschka and L. L. Iacono, "Vulnerable Cloud: SOAP Message Security Validation Revisited," in *IEEE International Conference on Web Services, 2009. ICWS 2009*, 2009, pp. 625–631.

[15] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, New York, NY, USA, 2011, pp. 3–14.

[16] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On Breaking SAML: Be Whoever You Want to Be," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2012, pp. 21–21.

[17] T. Jager and J. Somorovsky, "How to break XML encryption," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 413–422.

[18] M. Jensen, N. Gruschka, R. Herkenhoner, and N. Luttenberger, "SOA and Web Services: New Technologies, New Standards - New Attacks," in *Fifth European Conference on Web Services, 2007. ECOWS '07*, 2007, pp. 35–44.

[19] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Comput. Sci. - Res. Dev.*, vol. 24, no. 4, pp. 185–197, May 2009.

[20] K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, and P. Hallam-Baker, "Web services security: SOAP message security 1.1 (WS-security 2004)," *OASIS OASIS Stand. Feb*, 2006.

[21] K. Ballinger, D. Ehnebuske, C. Ferris, M. Gudgin, C. K. Liu, M. Nottingham, and P. Yendluri, "Basic profile version 1.1," *WS- Specif.*, vol. 8, pp. 1–1, 2004.

[22] N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," in *15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09*, 2009, pp. 301–306.

[23] R. Richards, "Simple API for XML (SAX)," in *Pro PHP XML and Web Services*, Apress, 2006, pp. 269–310.

[24] R. Richards, "Document Object Model (DOM)," in *Pro PHP XML and Web Services*, Apress, 2006, pp. 181–238.

[25] N. Bhalla and S. Kazerooni, "Web services vulnerabilities," *BlackHat Eur. Amst.*, 2007.

[26] "Web Service Security Overview, analysis and challenges." [Online]. Available: http://search.proquest.com/openview/971cc544f19f987fc4a5471c0fc1762f/1?pq-origsite=gscholar. [Accessed: 24-May-2015].

[27] C. Mainka, J. Somorovsky, and J. Schwenk, "Penetration Testing Tool for Web Services Security," in *2012 IEEE Eighth World Congress on Services (SERVICES)*, 2012, pp. 163–170.

[28] S. Kabeer, A. P. S, and V. D, "Infiltrate Testing Tool for Web Services Security," *IJRCCT*, vol. 2, no. 7, pp. 455–460, Jul. 2013.

[29] E. Tews, "Effective DoS attacks against Web Application Plattforms – #hashDoS [UPDATE3]," *Cryptanalysis - breaking news*. [Online]. Available: https://cryptanalysis.eu/blog/2011/12/28/effective-dos-attacks-against-web-application-plattforms-hashdos/. [Accessed: 25-May-2015].

[30] "Many more web platforms vulnerable to the hash collision attack (not only ASP.NET) #28C3 @hashDoS #hashDoS @ccc," *The Wiert Corner - irregular stream of stuff*. [Online]. Available: http://wiert.me/2011/12/29/many-more-web-platforms-vulnerable-to-the-hash-collision-attack-not-only-asp-net-28c3-hashdos-hashdos-ccc/. [Accessed: 25-May-2015].

[31] T. P. Chiem, "A study of penetration testing tools and approaches," Auckland University of Technology, 2014.

[32] "Soapui." [Online]. Available: www.Soapui.org. [Accessed: 24-May-2015].

[33] A. Falkenberg, C. Mainka, J. Somorovsky, and J. Schwenk, "A New Approach towards DoS Penetration Testing on Web Services," in *2013 IEEE 20th International Conference on Web Services (ICWS)*, 2013, pp. 491–498.

[34] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 1999, pp. 173–186.

[35] Z. Zheng and M. R. Lyu, "A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services," in *IEEE International Conference on Web Services, 2008. ICWS '08*, 2008, pp. 145–152.

[36] F. Curbera, Y. Goland, J. Klein, F. Leymann, S. Weerawarana, and others, "Business process execution language for web services, version 1.1," 2003.