A Large Scale Study of Web Service Vulnerabilities

Sushama Karumanchi* and Anna Squicciarini
Pennsylvania State University, University Park, Pennsylvania, 16802, United States
{sik5273, asquicciarini}@ist.psu.edu

Abstract

The pervasiveness of Web Services, compounded with seamless interoperability characteristics, introduces security concerns that are to be carefully considered with the envisioned internet architecture. In this paper, we propose a comprehensive study on Web Service vulnerabilities. We consider not only well known Web-based vulnerabilities such as SQL injection, session replay etc, but we also analyze Web-Service specific vulnerabilities and their potential of attacks due to poor service construction and lack of service maintenance. In our analysis, we classify each of the studied vulnerability according to a new taxonomy, discuss remedies and impact, and propose methods of detection based on real-time analysis. Our analysis is supported by the results of a large scale study involving over 2,000 real-world Web Services. Finally, we leverage our empirical finding by introducing a proxy-based solution that shields services and clients from any possible attacks.

Keywords: Web Service Vulnerabilities, Web service Security, Web service selection, Vulnerability Taxonomy, Classification

1 Introduction

The vision of the Internet involves building a new generation of applications made by composing services and data from different providers and organizations. This provides users with added-value services tailored to their needs. Towards the direction of achieving the vision of the Internet, Web services provide a simple interface between a provider and a consumer and are supported by a complex software infrastructure, which typically includes an application server, the operating system and a set of external systems (e.g., databases, payment gateways, etc). A Web Service includes several operations which can be remotely invoked by a client. Each operation is a method with several input parameters, and is described using a standardized XML format used to generate server and client code, and for configuration, the WSDL (Web Service Description Language). A broker is utilized to enable applications to find Web Services. Simple Object Access Protocol (SOAP) is used as a means of communication between the consumer and the provider.

However, Web-Services are subjected to several unique security concerns. These concerns do not apply in traditional distributed messaging techniques (e.g. RMI [21] and CORBA [10]) and are mainly due to the pervasiveness of Web-Services, compounded with seamless interoperability characteristics.

For instance, the SOAP protocol used for communication in Web Services does not address security by itself. A SOAP protocol can be bypassed by a firewall, and can get processed by the Web Service directly [22].

SOAP messages can also be easily exploited as shown in [22], simplifying the tasks of an attacker trying to compromise Web-Service communication.

Extensions to improve Web service security have been proposed by some recent works, e.g. [12, 18, 5, 6]. Also, there exist some standardization efforts for the security of Web services. WS-Security [20]

Journal of Internet Services and Information Security (JISIS), volume: 5, number: 1 (February 2015), pp. 53-69
*Corresponding author: College of Information Sciences and Technology, Pennsylvania State University, University Park, Pennsylvania, 16802, United States

for example, is a notable standardization effort, which is introduced to add security to SOAP messages by describing how the header part of the message can be used to pass along security information. However, it is not clear to what extent these initiatives have strengthened the overall security guarantees offered by the Web services. Taxonomies for generic software vulnerabilities have also been successfully proposed [27, 3]. Nevertheless, to the best of our knowledge, an in-depth study of Web Service vulnerabilities is missing. Most of the existing works point to vulnerabilities that are generic to any Web application, or focus on securing solely SOAP messages, but do not discuss vulnerabilities specifically related to Web Services, such as *WSDL vulnerabilities* [18].

In this paper, we define WSDL related vulnerabilities to be those that enable the attackers to know their existence through the analysis of the WSDL files of the services. Accordingly, we introduce a novel client-based classification of Web service vulnerabilities, as well as a proxy-based solution for efficient vulnerability detection. Our analysis emcompasses several Web service vulnerabilities. As mentioned, we not only consider known Web-based vulnerabilities such as SQL injection, session replay etc, but we also include vulnerabilities born specifically as a result of poor Web service construction and service maintenance, such as lack of encryption, invalid XML, parser attacks, and log file attacks. We also discuss the remedies and impact for each of the identified vulnerabilities, and propose methods of detection based on real-time analysis of the WSDL document describing the exposed Web Service. Through our classification, a client may be able to know which vulnerabilities can be quickly prevented by the client as well as which ones can be prevented without revealing any information to the service provider or the attacker.

As an extension to our previous work [13], we also introduce a proxy-based solution that may drastically improve the security guarantees offered by Web Services by shielding services and clients from any possible attacks. In our solution, the client does not evaluate the service for vulnerabilities nor do the service providers. Rather, a proxy performs this evaluation. Such a solution reduce the burden from the two parties, allowing them to concentrate on their specific tasks. The proxy analyses the candidate services searched by the client for any vulnerabilities, and accordingly suggests security settings for the client to employ, when making a selection of its desired composite service.

Our vulnerability analysis is supported by our empirical evaluation, which is carried out over 2,000 real-world Web services. We prototyped a vulnerability detector to evaluate these real-world Web services. Our evaluation indicates that many of the least studied vulnerabilities (in the context of Web services) are present in the wild such as *Password in Clear,* and *Invalid XML*. We provide a discussion on possible solution mechanisms and countermeasures.

The paper is organized as follows. In Section 2, we present some background on Web services. In Section 3, we present our taxonomy of services and discuss various vulnerabilities. We present our methodology employed to detect vulnerabilities and discuss the results in Section 4. We propose a proxy-based solution to enable the client to detect and avoid the vulnerabilities existing in the Web services in Section 5. We conclude in Section 6.

2 Web Services Standards and Security

In this section, we briefly overview some of the Web Services standards that are relevant for our study. Web Services Description Language (WSDL) is an XML format for describing Web services. WSDL describes the structure of a specific service using XML- formatted data. The information provided acts as an interface to the service, that is, the information includes service name and location, method names, argument types, return values and types. Through a WSDL description, a client application can determine explicit instructions on how to communicate with previously private applications, as well as determine the operations that are available to the consumer that he or she can invoke. WSDL files are typically

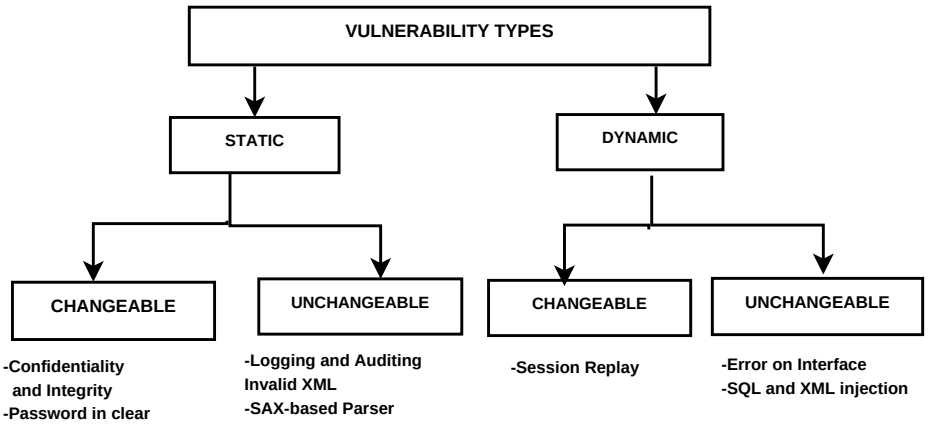

Figure 1: Taxonomy of Web Services Vulnerabilities.

stored in Web registries that can be searched by potential clients to locate Web Service implementations of desired capabilities. Due to exposed detailed methods and location information, several attacks can be crafted based upon vulnerabilities leveraging information in WSDL files. Essentially, the attacker, by analyzing the WSDL, is provided with critical information about various methods and parameters needed for the attack. These set of vulnerabilities are sometimes referred to as *WSDL vulnerabilities*. We note that, although some recent work has already acknowledged that Web services undergo WSDL threats [18, 8], we are not aware of any study of the impact of such vulnerabilities. Further, previous work has shown that Web Services are subjected to many other attacks, irrespective of the information stored in WSDL [22, 16, 17, 18, 2]. For instance, in Web applications, a client communicates with the application through a Web browser whereas in Web Services, the client directly interacts with the service. Hence, the Web Services are more vulnerable to attacks than traditional Web applications due to the absence of a browser in the middle of communication. Li et al. [14] propose an algorithm to test for vulnerabilities in Web services. They follow the approach of input parameter mutations and application of a combination of operators that would mutate the input provided to the Web service. However, they do not conduct a large scale study of Web service vulnerabilities, instead they test how their approach works. Our results provide the statistics of the types and the numbers of the vulnerabilities. They perform injection attacks in their approach. We not only consider injection attacks but also other attacks a Web service is vulnerable to.

In order to cope with these issues, Web Service security standards [20] are nowadays widely used to implement an end-to-end security solution between the sender and receiver in a SOA system. Digital signatures and message encryption are used within the WS-Security standard to ensure the confidentiality, non-repudiation and integrity of the messages. WS-Trust, WS-SecurityPolicy, WS-SecureConversation, WS-Federation and WS-Authorization protocols are additional security standards that augment the WS-Security specification.

WS-Security standards, however, cannot provide a comprehensive security solution to Web Services. For instance, XML-signatures can be used to protect the integrity of messages exchanged between the client and the service, but if the message sender itself is malicious, it can insert malicious content within the messages using its own genuine XML-signature [12].

Also, WS-Security is not effective to prevent many attacks. For instance despite the WS-Security standards, an attacker can launch XML injection, parameter tampering attacks [22], Denial-of-Service [12], and information disclosure attacks. We argue that security measures such as input validation and careful coding of Web Services are very important and complement WS-Security standards. These ob-

servations motivate us to undertake this study to find out how widely the Web Service vulnerabilities are spread out in the huge set of openly available Web Services.

3 A taxonomy of Web-Service Vulnerabilities

In this section, we introduce a new taxonomy for Web Services vulnerabilities, shown in Figure 1, and discuss few representative vulnerabilities for each introduced element in the taxonomy. We provide a novel client-centered perspective on the vulnerabilities Web Services currently face. Precisely, our approach is to take the viewpoint of a client to help it quickly detect and cope with the vulnerability being observed. The client-targeted classification has the following advantages: 1. The client knows which vulnerabilities can be prevented by the client (i.e., changeable), 2. The client knows which vulnerabilities can be *quickly* detected (i.e., static), and 3. The client knows which vulnerabilities can be prevented without revealing any information to the service provider or the attacker (i.e., static).

This viewpoint addresses a known shortcoming of previously proposed taxonomies, and of similar classification approaches based on software vulnerabilities [27, 3] which do not make their intended usage explicit.

For instance, a well known approach to classification of vulnerabilities [16, 17], is based on the nature of the software error originating the vulnerability or the security breach.

These classifications [16, 17] are interesting but general purpose. Furthermore, although very extensive, using the software error as a criteria may result in some degree of ambiguity, in that some vulnerabilities may arguably fall into two different categories. For example, if the criterion for classification is security breach, the well-known *SQL injection* vulnerability could fall into both *Tampering* and *Information Disclosure* vulnerability categories, since SQL exploits may result in tampering of the Web service as well as lead to unwanted information leakage. Also, we note that works that predict the occurrence of vulnerabilities exist [4]. However, in this work, we do not aim at predictive methods, but rather at designing strong taxonomies.

3.1 A Client-targeted Taxonomy

In order to develop a sound taxonomy, a main challenge is to identify unambiguous, orthogonal classification criteria for a set of objects [23]. Defining unambiguous classification criteria for software vulnerabilities is especially known to be non-trivial [4]. In this work, the vulnerabilities are classified based on the following: (1) by the detection method and (2) by checking if the client , by itself, can prevent the attack related to the vulnerability. Accordingly, we define two mutually exclusive categories of vulnerabilities: *Static* and *Dynamic* vulnerabilities. *Static* vulnerabilities can be detected without the execution of a service whereas *dynamic vulnerabilities* can be detected upon execution.

In order to detect *dynamic vulnerabilities*, the client needs to receive a response from the service, whereas, for the detection of static vulnerabilities, there is no need of any feedback or response from the service. To detect a *static vulnerability*, the client can utilize the available resource belonging to the service (e.g., WSDL) and does not need feedback or response from the service. Each of these two classes of vulnerabilities are further classified into *Changeable* and *Unchangeable*. *Changeable* vulnerabilities can be prevented by the client and the service on the fly when a client makes a call to the service. That is, the vulnerability can be addressed without modifying the core functions or the business logic of the Web service itself. For instance, the client may modify its original input or add to its original input in order to prevent vulnerabilities. For example, a client protects its password by encrypting it with a cryptographic key when the client finds that the service does not provide any mechanism for protecting the password. *Unchangeable* vulnerabilities can only be prevented if the service undergoes

some architectural and structural change. That is, the service provider needs to modify the service in order to prevent vulnerabilities which cannot be done dynamically when a client call the service. For example, if the service is prone to generate an error on the interface on execution, it cannot be prevented either by the client or the provider during the time of service execution by the client.

Note that some vulnerabilities classified as changeable may, of course, be addressed by the server which could change the WSDL or its applications to address them. Our approach is to label vulnerabilities as changeable if the client has the option to prevent the vulnerability even if the server could theoretically also address them directly.

In Figure 1, we report examples of vulnerabilities classified under our taxonomy. We note that our taxonomy is orthogonal to existing classifications, as we provide a client-targeted taxonomy.

3.2 Vulnerabilities

Next, we discuss some vulnerabilities for each of the classes we have identified. We select vulnerabilities covering a broad and diverse set of representative software and architectural problems. Our discussion focuses on Web service-specific vulnerabilities, whose corresponding attacks mainly exploit information stored in WSDL files. Our empirical evaluation demonstrates that most of the WSDL vulnerabilities are often overlooked, and yet exist in the wild and are poorly protected.

We note that our discussion focuses on specific instances of selected vulnerabilities. Multiple variations for every discussed vulnerability may exist, which however have the same inherent nature and similar exploit methods. For instance, for the parsing vulnerability (V2 in the description below), there exist different variations such as XML Bomb, huge file size, SOAP array attack vulnerabilities, etc. In all of these instances, the basic idea is the same: the XML file is modified to make the processing time of the file huge.

3.2.1 Static Vulnerabilities

We now present examples of static vulnerabilities. We discuss several vulnerabilities that are WSDL-related (V1-V4), and provide an example of a non-WSDL-related vulnerability (V5).

V1. Password in Clear

Purpose: A service requires a password from its client to authenticate the client. A Web Service requests the client for a user name and password by creating an authentication method among its other methods related to the service. Just like any other method of the service, the interface of an authentication method is published in the WSDL. The client makes a call to this method by providing his user name and password, after which the service authenticates the user. The following is a code snippet showing a login method in terms of an operation.

```
<message name="LoginInput">
    <part name="body" element="xsd1:LoginRequest"/>
  </message>
<message name="LoginOutput">
    <part name="body" element="xsd1:LoginResponse"/>
  </message>
  <portType name="LoginPortType">
    <operation name="Login">
      <input message="tns:LoginInput"/>
      <output message="tns:LoginOutput"/>
```

```
    </operation>
  </portType>
```

Vulnerability: A service is said to have a *Password in Clear* vulnerability when it does not use password encryption methods to protect the password at the message level. Even though the client and the service employ a transport level security protocol such as TLS or SSL, the password is still at a message-level threat. Message-level security is very important in cases where there exist intermediate nodes receiving the client's message or request. For instance, consider a service requesting the client for its password. The client sends the password in clear as an argument to the authentication method. However, the client method call might pass through multiple intermediate nodes or servers which are not necessarily authorized to access the client's password. In this case, even though the client method call is encrypted using a transport layer protocol, after receiving the client call, the intermediate nodes can access the client's password.
Remedy: The *Password in Clear* vulnerability can be resolved by defining policies in the WSDL of a service. WS-SecurityPolicy standard along with the WS-Policy standard can be used to define security policies within the WSDL to request the client to encrypt its password. This vulnerability is classified as *Changeable* as the client can encrypt its password with the service's public key if it exists even when there is no policy defined on the password. The following is an example of an encryption policy defined in a WSDL file asking the client to encrypt its input message, in which case, the client encrypts its password.

```
<wsp:Policy wsu:Id="InputIdentifier">
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:EncryptedParts>
                <sp:Body/>
            </sp:EncryptedParts>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

V2. Invalid Parser
Purpose: A parser is required at the service end in order to parse the service requests of the client.
Vulnerability: A service is said to have a *Parser* vulnerability when proper validation techniques are not in place at the service end during parsing of the service requests. Two types of parsers are commonly used in the context of Web Services: DOM-based parser and SAX-based parser [26]. DOM-based parser is prone to denial of service attacks. This is mainly because DOM-based parser places whole of the XML request data in the memory for parsing. One such dangerous attack is the *XML Bomb* attack, where an attacker writes an XML file with huge number of nested elements or entities. Due to this attack, the parser allocates large memory and it is stuck indefinitely parsing the huge number of elements, leading to denial of service to other clients requesting the service. Also, the following attacks can take place: inputting large number of files for parsing, malformed XML (e.g., unclosed tags), malicious attachment, soap array attack (huge number of XML elements), and large XML document size. SAX-based parser is more prone to XML injection attacks, where an attacker inputs data to the service which can query data in unauthorized mode. Though both DOM and SAX based parsers are vulnerable to denial of service and XML injection attacks, the above attacks are more critical to each of the parsers. This vulnerability can be detected at the service provider end. Though *Parser* vulnerability is not explicitly dependent on the WSDL, it is considered to be partially dependent on the WSDL as the attacker can carefully frame

the XML input which is logically correct according to the schema defined or referred to in the WSDL, but which is a malicious one.
Remedy: The *Denial of Service* attack in terms of *XML Bomb* attack can be resolved by validating the size of input stream when an XML request arrives at the service end. The *XML Injection* attack can be resolved by properly validating the input from the client by defining a proper XML Schema. This vulnerability is classified as *Unchangeable* as the code and the type of parsers cannot be changed during the client's call to the service.

V3. Invalid XML
Purpose: Validation of an XML file is needed in order to prevent attacks that submit an XML file with malicious content or XML file with wrong data types.

Vulnerability: A service is said to have an *Invalid XML* vulnerability when the schema related to the service is defined within the WSDL file of the service. The following is an example of a schema defined within the WSDL (the other parts of the file such as messages, port types, etc, are not shown):

```
<wsdl:types>
    <s:schema elementFormDefault="qualified"
          targetNamespace="http://test.org/">
      <s:element name="WeatherRequest">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
              name="City" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1"
             name="Zipcode" type="s:int"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="WeatherResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
               name="Resp" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
</wsdl:types>
```

Exploiting this vulnerability, an attacker can modify the actual schema with a different schema and replace it in the WSDL file. This attack is called *Invalid XML or Schema Poisoning* attack, wherein when the client accesses the WSDL file, it is prompted with an *Invalid XML* error due to which the client would not be able to use the WSDL file to make function calls to the service.

For instance, in the above schema, the attacker might modify the above inline schema by replacing the `string` type with an `int` type. Hence, the client is forced to provide an integer instead of a string due to which the client would get an *Invalid XML* error every time it calls the service.

Remedy: The *Invalid XML* vulnerability can be resolved by defining the schema outside of the WSDL file. The attacker will not have access to the schema for modification. This vulnerability is classified as *Unchangeable* as the service provider defines the schema beforehand in the WSDL and cannot be changed when the client makes a call to the service.

V4. Confidentiality and Integrity
Purpose: A client needs to protect its data during transit from unauthorized sniffing and modification from attackers if the data is important and sensitive.
Vulnerability: A service is said to have a *Confidentiality and Integrity* vulnerability when it does not employ any kind of encryption or integrity techniques over its input and output data. When the input data sent to the methods by the client and when the output data from the service are not encrypted, there is a breach to the confidentiality of the client's data in transit similar to the *Password in Clear* vulnerability. However, *Password in Clear* vulnerability is present only in the `operations` element of the WSDL file and deals only with the password, whereas *Confidentiality and Integrity* vulnerability is concerned to any data present within any element within the WSDL file. Also, the client's data in transit to the service might be altered by attackers in which case the integrity of the client's data is lost. This vulnerability can be detected by analyzing the WSDL file of the service by checking if there are any security policies defined in the WSDL file regarding the encryption and integrity of the messages or data.
Remedy: The *Confidentiality* vulnerability for input data can be resolved by requiring the client to encrypt its data during transit. This requirement can be achieved by defining XML encryption policies using WS-SecurityPolicy standard along with the WS-Policy standard within the WSDL. For confidentiality of output data, the service needs to employ the encryption methods too. The *Integrity* vulnerability can be resolved by requiring the client to use digital signatures over its input data. The service can achieve this by defining XML digital signature policies using WS-SecurityPolicy standard along with the WS-Policy standard within the WSDL. This vulnerability is classified as *Changeable* as the client, before making a call to the service, can encrypt the message using the public key of the service to ensure confidentiality, and the client can sign the message using its private key to ensure integrity.

V5. Logging

Several static vulnerabilities that are not WSDL-related exist. For instance vulnerabilities such as the *Logging* vulnerability and *Services in Public Business Registries* vulnerability arise irrespective of WSDL data.
Purpose: A service provider needs to log or store critical activities that take place with respect to the service for security reasons. For instance, a service logs the login time and login username.A service also needs to audit the stored log files for security reasons. When a failed login occurs, by auditing the log file, the service provider can investigate about a probable attack.
Vulnerability: A service is said to have a logging vulnerability when it is prone to log injection. That is, the service logging mechanism does not properly validate the input of the service. In a log injection attack, an attacker injects his own phrase into the service provider's log file. The attacker injects a targeted phrase in the input he provides to the service. For instance, consider a service taking a user name of the user as one of its inputs. The following is a normal log file generated after a genuine user User1 invokes the service.

```
Successful login attempt for User1.
```

An attacker's intention could be to defame User3. The attacker provides his input as "User2. [LINEBREAK] Failed login attempt for User3. [LINEBREAK] Failed login attempt for User3. [LINEBREAK]

Type of Service	Authenticating Services		Non-Authenticating Services		% of Services Authenticating
	No of Services	Avg. no of Operations	No of Services	Avg. no of Operations	
Business	38	9	27	5	58
Location	41	9	71	6	37
Communication/Entertainment	29	15	36	8	44.6
Scientific/Security	15	3	695	23	2
Search	5	7	50	8	9
Other	224	15	1115	7	17
Total	352	66	1994	62	

Table 1: Authenticating and Non-Authenticating Services

Failed login attempt for User3. [LINEBREAK] Failed login attempt for User3. [LINEBREAK] Failed login attempt for User3.", so that the following is written to the log file.

```
Failed login attempt for User2.
Failed login attempt for User3.
Failed login attempt for User3.
Failed login attempt for User3.
Failed login attempt for User3.
```

Since the service logging mechanism does not properly validate the input of the service, especially, the linebreak in the example above, the attacker is able to inject into the log file. User3 is mistakenly suspected for his login behavior due to this logging vulnerability.
Remedy: The *Logging* vulnerability can be resolved by properly validating the inputs provided by the user. The logging validation mechanism by avoiding meta-characters such as linebreak, separators, etc. This vulnerability is classified as *Unchangeable* as the log files are at the service provider's end.

3.2.2 Dynamic Vulnerabilities

Dynamic Vulnerabilities (i.e. can only be detected upon the execution of a service) are widely present in generic Web applications and have been studied [11, 19] more in depth than the *Static Vulnerabilities*. We now provide a high level review of the most seen ones in Web services.

V6. Error on Interface
Purpose: Errors are used to let the developer know to alter the code if there is a bug in the code, and to let the user of the service know to alter his or her input to the service.
Vulnerability: A Web Service is said to have an *Error on Interface* vulnerability when it throws an error on the client's user interface or browser when the client makes a call to the service which might reveal to an attacker the internal details of the service such as their secret directory information, database dumps, and stack traces. For example, the following code of a service reveals the path information to the service invoker:

```
char* path = getenv("PATH");
...
```

Type of Service	Total Services	Pwd in Clear	Conf and Int	Invalid XML	Total Vulnerabilities
Business	65	37	64	63 (96.92%)	164
Location	112	41	112	110 (98.21%)	263
Communication/Entertainment	65	29	65	61 (93.8%)	131
Scientific/Security	710	15	710	686 (96.62%)	1411
Search	55	5	55	52 (94.54%)	112
Other	1339	224	1339	1065 (79.53%)	2748
Total	2346	351	2345	2157(91.94%)	4853

Table 2: Static Vulnerabilities by Service Type

```
sprintf(stderr, " No file found on path %s n", path);
```

Remedy: The *Error on Interface* vulnerability can be avoided by handling error messages in the service code. This vulnerability is classified as *Unchangeable* as the service provider cannot change the service code while the client makes a call to the service.

V7. SQL and XPath Injection

Purpose: The inputs of the clients are transformed into SQL or XPath queries. The queries are used to query the SQL and XPath databases of the services.

Vulnerability: A Web Service is said to have an *SQL or XPath Injection* vulnerability when an attacker can input hidden queries in his or her XML requests to retrieve data from the database of the service. Similar to SQL injection attack, XPath injection attack takes place when the service is using XML documents to store user data instead of an SQL database.

Remedy: The *SQL and XPath Injection* can be avoided by defining an XML schema that carefully validates all possible types of inputs from the user. This vulnerability is *Unchangeable*.

V8. Session Replay

Purpose: Session is maintained between a client and a service, so that the client need not repeat providing the same data to the service for consecutive method calls. For instance, in order to authenticate a client, the service requires the user name and password from the client every time it calls the service. Hence, by maintaining a session with the client, the service need not authenticate the client multiple times.

Vulnerability: A service is said to have a *Session Replay* vulnerability when it maintains sessions through session IDs. An attacker can get hold of the session ID and reuse it to gain unauthorized control of the session of an authorized client by sending a request to the service using the session ID.

Remedy: *Session Replay* vulnerability can be resolved by the service and the client using a nonce during communication, with the client, that involves session ID. This vulnerability is classified as *Changeable* as the client can send a nonce along with the session ID when it makes a call to the service.

4 Empirical Evaluation

To assess the extent to which Web Services vulnerabilities are an actual problem in today's Internet, we tested 2346 real Web Services. These services are taken from the Web Service collection of Al-Masri and Mahmoud [9], who obtained them from UDDI Business Registries and the World Wide Web. In total, the authors [9] collected 2507 services. Out of this dataset, we disregard 160 services which generated

parser errors. The parser was unable to find the schema files located in different locations, that is, outside of the WSDL file.

4.1 Methodology

We developed a WSDL parser in order to efficiently detect static vulnerabilities. Each WSDL file belonging to the 2507 services is associated with a service. The parser reads all the WSDL files in a loop and processes each WSDL. It specifically detects static vulnerabilities by looking into certain elements and information within the WSDL. It first looks for the service name by parsing the file utilizing a `Definition` object. Then, based on the service keywords (extracted from the WSDL and compared with Wordnet dictionary to ensure semantic relevance with the category), the parser classifies the service into a type. The parser compares the service name with a set of words which are commonly used for a category. For instance, to detect if the service falls into business category (discussed in Section 4.2), the parser compares the service name with the following words: *order, business, price, purchase, rate, quote, accounting, stock, tax, market, finance, etc*. More details on the type-based organization of the dataset used for our tests are provided in the next section.

Next, it checks for the elements defining the vulnerabilities. For instance, for the *Password in Clear* vulnerability, our parser looks for the term `Password` in the WSDL document within the `operations` elements. The parser also looks if there are any policies defined by looking if any of the `wsp` or `wsap` elements (see Section 3.2.1, V1) exist in the WSDL file. If both these cases are true, then the parser warns of a *Password in Clear* vulnerability. Similarly, the parser looks for the presence of `wsp` or `wsap` elements in the WSDL for detecting the *Confidentiality and Integrity* vulnerability. However, for detecting *Password in Clear* vulnerability, the parser, in addition to checking for the presence of policies in the WSDL file, looks for authentication methods, and within them looks for the argument "Password", whereas, for detecting *Confidentiality and Integrity* vulnerability, the parser just checks if any policies are defined within the WSDL file. For both of these vulnerabilities, checking for a `wsp` or `wsap` element enables us to know whether usage of encryption or integrity technologies are mandated by the service.

In order to detect whether *Invalid XML* vulnerability exists, the parser looks for the presence of a schema definition element within the `types` element in the WSDL file. If the schema is defined in the WSDL file, the parser concludes the presence of the *Invalid XML* vulnerability. The *Parser* and *Logging and Auditing* vulnerabilities cannot be detected by us as the parser type information is available at the service end, and the log and audit information is also available at the service provider. However, if the client is able to negotiate about this information with the service, then these vulnerabilities can be identified.

To test the dynamic vulnerabilities, we utilized a commercial Web Service vulnerability scanner called Acunetix Web vulnerability scanner [1]. We chose this scanner as it is well known for Web Services vulnerability detection [25] and can detect the most popular Web-based vulnerabilities (including Error on Interface, and SQL and XPath Injection) in a reliable fashion.

4.2 Results

As Web services can be of disparate types and can have varying degree of complexity, we organized the WSDL files of the dataset in different groups, based on the Web Service's types. We anticipate that Web services belonging to the same type will have a very similar set of functionality and corresponding architecture, and therefore may be prone to a similar set of vulnerabilities.

We classified the services into 6 types based on their provisioned service: *Business* (eg., quote retrieval), *Location* (eg., weather), *Communication* and *Entertainment* (eg., email, travel, holiday), *Scientific/Security* (eg., gene variations, encryption), *Search* (eg., search for university data), and *Others*.

These categories reflect the common types of Web services exposed in public registries.

In Table 1, we show the average number of operations for each service type. As reported, *Scientific/Security* services expose a large number of operations, followed by *Communication* and *Other* service types. We note that services in the *Business* category have higher percent of authenticating services, followed by *Communication* services. In the table, we further distinguish among authenticating (e.g. request a password to access) and non-authenticating services.

We observe that the authenticating services have an average of 9.4 operations or functions in their services whereas the non-authenticating services have an average of 8.8 operations. Intuitively, this is because authenticating services are more complex than the non-authenticating services and usually offer complex and possibly sensitive operations.

Table 2 shows the breakdown of number of vulnerabilities detected by service type. As can be quickly observed, there is an extremely large number of vulnerabilities identified per service type. Any vulnerability present in each service occurs at least once. First, we note that non-authenticating services have more static vulnerabilities (96.4%) compared to authenticating services (94.5%). Non-authenticating services are typically services with simpler communication protocols, that require limited interaction and storage of sensitive data with end users. Hence, they probably do not have a secure architecture compared to authenticating services.

Further, as shown in Table 2, almost all (96.6%) of the tested services do not specify any policies and hence they suffer from *Confidentiality and Integrity* vulnerabilities. A similar result is obtained for the *Password in Clear* vulnerability which is widely present among the services that request a password from the client. We find that all the authenticating services, irrespective of the service type have a *Password in Clear* vulnerability.

We reason about the absence of message level confidentiality and integrity protection by the services as follows: 1. Confidentiality and Integrity of data can be achieved at the transport layer level with the use of HTTPS, and 2. The tested services are publicly available and are not related to internal processes of a business and hence, might not have maintained message-level confidentiality and integrity of services. Message-level security is very important when the client message path includes multiple other applications or services which might be connected with different, possibly non-secure transport protocols. In such a case, transport level security is not sufficient to protect the messages. Hence, we consider addressing the *Confidentiality and Integrity* and *Password in Clear* vulnerabilities very critical in the Web Service life-cycle.

Also, we note that the large majority of the services define their schema within the WSDL itself. As discussed in Section 3.2.1, exposing the schema leads to *Invalid XML* vulnerability. We note that this vulnerability is mostly observed in Location Services and Search Web-services. The services from the *Other* category are least vulnerable (79.5 %) to *Invalid XML* attacks when compared to the remaining service types. This is quite unexpected, and probably identifies that there is a great variety of Web-services which are more carefully architected than others.

In summary, from the above, we see that static vulnerabilities are alarmingly common in publicly exposed Web Services, with *Confidentiality and Integrity* vulnerability being the most common. Confidentiality and integrity is considered a most important vulnerability, as previous research [7] has shown that in spite of the presence of cryptographic protocols such as SSL/TLS, breaches of confidentiality and integrity of data are likely.

In regard to dynamic vulnerabilities, we randomly selected 300 services and checked whether they had any dynamic vulnerabilities. We selected proportionally equal number of services from all service types. We specifically tested for *Error on Interface* and *SQL Injection* vulnerabilities as these are critical vulnerabilities that could dramatically affect Web Services, as discussed in Section 3.2.2. We found very few vulnerabilities. Interestingly, the tested Web Services have either 0 or multiple vulnerabilities. Precisely, 7% of the services have at least one vulnerability (i.e. 6 services), but all of such services have

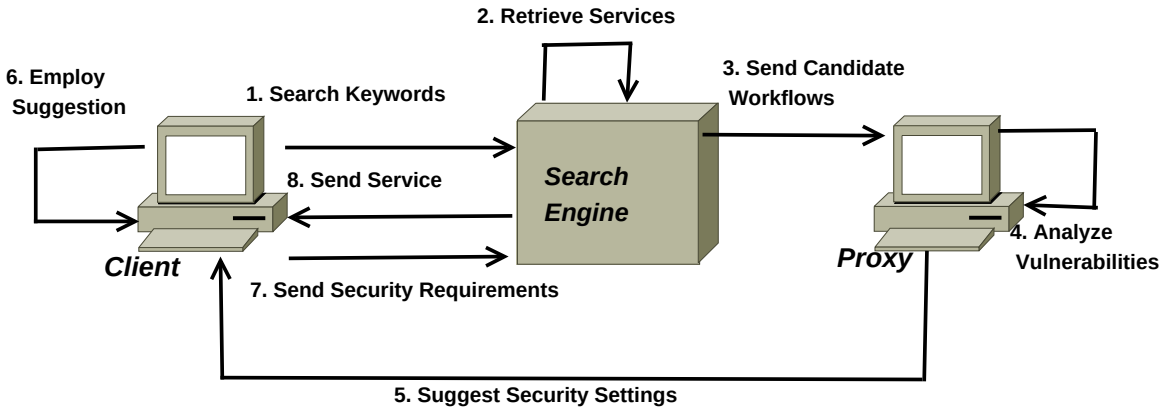

Figure 2: Proxy-based Solution

multiple occurences of the same dynamic vulnerabilities (14 each, on average). We speculate that these are instances of poorly maintained services, and therefore, when the dynamic vulnerabilities exist in a service, they affect multiple service operations leading to major security holes.

5 A Proxy-based Solution for Efficient Vulnerability Detection

Our tests revealed an alarming presence of static vulnerabilities in publicly exposed services, highlighting the need of a solution that prevents the attacks that exploit these vulnerabilities. Several solutions can be devised. A first intuitive approach would be a server-side solution, that involves requesting the Web Services to modify their WSDL file or code upon detecting the potential security issues. This approach might address most of the discussed vulnerabilities, but it is not feasible as the client would have to wait while the service modifies its WSDL file or code, whereas service invocation is dynamic.

A client-side solution that involves automatic provision of all possible security measures is a possible alternative to the server-based approach. In this case, a client can employ mechanisms such as message encryption between the client and the service (addressing confidentiality and password-in-clear vulnerabilities), and adding a nonce to the communication (addressing session replay vulnerability). However, this approach imposes a great computational burden to the client. As security methods are costly in terms of computation, this type of solution that automatically employs all security measures does not scale well.

Finally, a "middle-ground" solution, that combines above approaches while limiting their shortcomings is a proxy-based solution. This approach is based on the idea of suggesting security settings according to the vulnerabilities of the candidate services, prior to Web Services selection. The extent to which the service is to be secured is a decision left to the client (as in the client-based approach), that can select the preferred modifications and improvements prior to selection. In this way the dynamic invocation requirement of the service would be met, only client-chosen security setting modifications would be required. From our empirical analysis, we note that the two highly occurring vulnerabilities, *Password in Clear* and *Confidentiality and Integrity*, are *Changeable* vulnerabilities. Hence, using the proxy-based solution would enable the client to easily avoid attacks arising from such popular vulnerabilities.

We now briefly present the proxy-based approach (Figure 2), that can be used for both single Web Services and for composite services (i.e. multiple services combined to provide a single service). A step-by-step presentation of the service retrieval by the client and of the proxy server is as follows:

1. The client requests for a service by providing search terms to the Web Service search engine.

2. All possible services are retrieved by the search engine based on the client's search terms or input [24].

3. Each work flow activity has a set of candidate services [15].

4. Before providing the service to the client, the proxy server analyses all the candidate services in each activity to look for any security vulnerabilities.

5. The proxy suggests security settings to the requesting client.

6. The client considers these security settings and employs these settings or requests the search engine to select a composite service based on the suggested settings.

7. The search engine selects a combination of services according to the security setting requirements of the client if any.

8. The service composer (not shown in the figure; used for composing multiple services) within the search engine executes the selected composite service.

9. The client receives the output.

We further elaborate on each of these steps by means of a simple use-case. The search engine retrieves multiple combinations of candidate services corresponding to a workflow. For instance, the following is a workflow of activities of a composite "goods service": {`Goods Inquiry`, `Goods Supply`, `Goods Purchase`}. One of the corresponding candidate composite services is: `InquireGoodsServ`, `SupplyServ`, `PaymentServ`, where `InquireGoodsServ`,`SupplyServ`, and `PayServ` are service names.

The proxy, upon receiving the candidate services for the workflow activity, detects vulnerabilities over each candidate composite service, using WSDL parsers and scanners.

For each activity, the proxy records two sets, the *type of vulnerabilities* {*Type1, Type2,...,TypeN*} and the *number of vulnerabilities* {*numType1, numType2,...,numTypeN*} for all services in an activity. The proxy suggests the client only those security settings corresponding to vulnerability types that are dominating, and informs about non-changeable vulnerabilities for which the client may have to either accept as is or refuse the composition. That is, the proxy selects the vulnerability types which cross a threshold T wherein $T \in [0,1]$ is computed according to the likelihood of the occurrence of the vulnerability being exploited and its possible impact. For instance, consider the following sets of *vulnerability type* and *vulnerability number* for an activity

$A1$: {*Password in clear, Invalid XML, Invalid Parser*} and {*0.1, 0.2, 0.02*} and let the threshold T be 0.1. The proxy suggests security settings that correspond to dominating vulnerabilities, that is, *Password in Clear* and *Invalid XML* to the client. That is, the proxy suggests to encrypt the client's password and to pose a requirement on the composer to select services that have their schema defined outside of their WSDL file. Similarly, for each activity in the workflow of a composite service, the proxy suggests the client to deploy one or more security settings. The client has a choice to select some or all these security settings and accordingly deploy them, or place them in its request to the composer for selection of services. As an alternative, the proxy may deploy the security settings by modifying the client messages or by placing them in the request to the composer. The composer makes a selection of a composite service according to the client's security requirements if any.

In summary, using a proxy server for analyzing the candidate services and suggesting security settings to the user is effective due to the following reasons: (1) The proxy can help protect the user from being exposed to attacks as the user can change his or her security settings (2) The proxy does not require the service provider to alter anything at its end, so the security is provided to the client on the fly. (3) The

Proxy may have greater access to both the Server and Client and be able to address server-side problems, such as auditing and logging issues. The limitation of this approach is the fact that the proxy adds an extra layer of communication. However, the extra layer can be avoided by building the proxy into the search engine itself.

6 Conclusion

As Web Services technologies become an important component of the Internet vision, we urge a better understanding of their security guarantees. Toward meeting this goal, in this paper we introduced a novel simple taxonomy to classify Web Services vulnerabilities. Within the provided classification, we discussed various vulnerabilities associated with Web Services. To verify how common these vulnerabilities are, we analyzed vulnerabilities of over 2000 real-world Web Services. Our experiments show that there are a huge number of vulnerabilities in the publicly available services, which call for a comprehensive solution to prevent the exploitation of these vulnerabilities. We suggest the adoption of a proxy-based solution to counter these vulnerabilities. As part of the future work, we plan to complete the deployment and testing of the proposed proxy-based solution.

Acknowledgement

This material is partially based upon work supported by the National Science Foundation under Grant No. #1250319.

References

[1] Acunetix Web Application Security. http://www.acunetix.com/.

[2] N. Antunes and M. Vieira. Evaluating and improving penetration testing in web services. In *Proc. of the 23rd International Symposium on Software Reliability Engineering (ISSRE'12), Dallas, USA*, pages 201–210. IEEE, November 2012.

[3] T. Aslam. *A taxonomy of security faults in the unix operating system*. PhD thesis, Purdue University, 1995.

[4] C. V. Berghe, J. Riordan, and F. Piessens. A Vulnerability Taxonomy Methodology applied to Web Services. IBM Technical Paper, 2005.

[5] E. Bertino, L. Martino, F. Paci, and A. Squicciarini. *Security for Web Services and Service-Oriented Architectures*. Springer, 2009.

[6] M. B. Brahim, T. Chaari, M. B. Jemaa, and M. Jmaiel. Semantic matching of ws-securitypolicy assertions. In *Revised Selected Papers from the 10 th Internation Conference on Service-Oriented Computing Workshops (ICSOC'12), Cyprus, Paphos, LNCS*, volume 7221, pages 114–130. Springer Berlin Heidelberg, December 2012.

[7] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In *Proc. of the 23rd Annual International Cryptology Conference (CRYPTO'03), Santa Barbara, California, USA, LNCS*, volume 2729, pages 583–599. Springer Berlin Heidelberg, 2003.

[8] Y. Demchenko, L. Gommans, C. De Laat, and B. Oudenaarde. Web services and grid security vulnerabilities and threats analysis and model. In *Proc. of the 6th IEEE/ACM International Workshop on Grid Computing, Seattle, Washington, USA*. IEEE/ACM, November 2005.

[9] A.-M. E. and H. Mahmoud. Web Service Data Set. http://www.uoguelph.ca/ qmahmoud/qws/dataset/.

[10] O. M. Group. CORBA. http://www.corba.org.

[11] W. G. Halfond, J. Viegas, and A. Orso. A classification of SQL-Injection attacks and countermeasures. In *Proc. of the 2006 IEEE International Symposium on Secure Software Engineering (ISSSE'06), Washington, DC, USA*.

[12] J. Holgersson and E. Soderstrom. Web service security - vulnerabilities and threats within the context of ws-security. In *Proc. of the 4 th International Conference on Standardization and Innovation in Information Technology (SIITT'05), Geneva, Switzerland*, pages 138–146, September 2005.

[13] S. Karumanchi and A. Squicciarini. In the wild: A large scale study of web service vulnerabilities. In *Proc. of the 29th Symposium on Applied Computing (SAC'14), Gyeongju, South Korea*, pages 1239–1246. ACM, March 2014.

[14] Q. Li, J. Chen, Y. Zhan, C. Mao, and H. Wang. Combinatorial mutation approach to web service vulnerability testing based on soap message mutations. In *Proc. of the 9th International Conference on e-Business Engineering (ICEBE'12), Hangzhou, China*, pages 156–162. IEEE, September 2012.

[15] Y. Li and C. Lin. QoS-aware service composition for workflow-based data-intensive applications. In *Proc. of International on Web Services (ICWS'11), Washington DC, USA*, pages 452–459. IEEE, July 2011.

[16] Microsoft. STRIDE Categories. http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx.

[17] Microsoft. Vulnerability Categories. http://msdn.microsoft.com/en-us/library/aa302418.aspx.

[18] A. Mirtalebi and M. R. Khayyambashi. Enhancing security of web service against wsdl threats. In *Proc. of the 2nd International Conference on Emergency Management and Management Sciences (ICEMMS'11), Beijing, China*, pages 920–923. IEEE, August 2011.

[19] E. Moradian and A. Hakansson. Possible attacks on xml web services. 6(1B), January 2006.

[20] OASIS: WS-Security 1.1. http://www.oasis-open.org/specs/.

[21] Oracle. RMI. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[22] N. Sidharth and J. Liu. Intrusion resistant soap messaging with iapf. In *Proc. of the 2008 Asia-Pacific Services Computing Conference (APSCC'08), Washington DC, USA*, pages 856–862. IEEE, December 2008.

[23] G. G. Simpson. *Principles of Animal Taxonomy*, volume 20. Columbia University Press, 1961.

[24] H. Song, D. Cheng, A. Messer, and S. Kalasapur. Web service discovery using general-purpose search engines. In *Proc. of the 2007 International Conference on Web Services (ICWS'07), Salt Lake City, USA*, pages 265–271. IEEE, July 2007.

[25] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Proc. of the 39 th International Conference on Dependable Systems Networks (DSN'09), Estoril, Portugal*, pages 566–571. IEEE, June 2009.

[26] W. Wang. Security based heuristic sax for xml parsing. In *Proc. of the 2007 International Conference on Security and Management (SAM'07), Las Vegas, USA*, pages 179–185. CSREA Press, June 2007.

[27] S. Weber, P. A. Karger, and A. Paradkar. A software flaw taxonomy: aiming tools at security. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

Author Biography

Sushama Karumanchi is a Ph.D. candidate at the College of Information Sciences and Technology, at the Pennsylvania State University. Her research interests include security and privacy issues in distributed systems and computer networks. Karumanchi earned a Master of Science in Computer Science from the University of Kentucky in 2010.

Anna Cinzia Squicciarini is an assistant professor at the College of Information Sciences and Technology, at the Pennsylvania State University. During the years of 2006-2007 she was a post doctoral research associate at Purdue University. Squicciarini's main interests include access control for distributed systems, privacy, security for Web 2.0 technologies and cloud computing. Squicciarini earned her Ph.D. in Computer Science from the University of Milan, Italy, in 2006. Squicciarini is the author or co-author of more than 65 articles published in refereed journals, and in proceedings of international conferences and symposia.