Software Vulnerability Analysis for Web Services Software Systems

Weider D. Yu* Dhanya Aravind Passarawarin Supthaweesuk
Computer Engineering Department, San Jose State University
San Jose, California, USA 95192-0180
*Email: Weider.Yu@sjsu.edu

Abstract

The use of Web Services has begun to significantly impact organizations and companies. Major business oriented objectives in reducing costs, shortening time, and improving quality and productivity can be achieved by using the Web Services technology. The Web Services software technology enables software components independently developed in disparate platforms to interact and collaborate in a seamless manner. They constitute a loosely-coupled, distributed system that is highly scalable. However, they also inherit the potential vulnerabilities of such systems. As Web Services increase in complexity and connectivity, the associated security risks also increase exponentially. Many of the security breaches can be traced back to poor verification and validation tasks. In this paper, a study on security related software vulnerabilities in SOAP-based Web Services is presented. The security context of traditional Web applications is compared to that of Web Services. An attempt has been made to map common attack patterns to security verification requirements with regard to Web Service software systems.

1. Introduction

Web Services are self-contained, modular software applications that can be described, published, located, and invoked over a network, generally, the World Wide Web. Web Services technology provides a cost-effective way to integrate applications developed in disparate platforms. It is predicted that it can bring down the system integration cost by 20 percent to speed up the adoption of Web Services in the future.

However, the enhanced connectivity and flexibility attributed to Web Services comes at the cost of increased security risks. According to Gartner Group, XML Web Services will reopen 70% of the attack paths closed by firewalls over the past decade [11].

Web Applications, forerunners of Web Services, have suffered from a multitude of exploits even with the application of the basic principles of security authentication, authorization, and auditing [1]. However, hackers and other malicious users are uncovering new techniques at the SOAP/XML data-level that bypass HTML and target weaknesses in Web Services programming, technology, and architecture [2]. For example, the Web Service Definition Language (WSDL) documents [6] used to describe Web Services are a technological weakness because they may be used as a “guide book” for vulnerabilities. Using this information, a security breach can penetrate deeply into the business logic, and ultimately into back-office systems [13].

Currently, the number of vulnerabilities specific to Web Services reported to public database is low. However, the number will increase rapidly with the growth of Web Services. In this scenario, it is worthwhile to study from the vulnerabilities reported for Web Applications, whose vulnerability domain includes that of Web Services. In this paper, an attempt is made to map the Web Application vulnerabilities and attack patterns to those of Web Services. A software vulnerability fault model for Web Services is developed to analyze the attack targets and based on the fault model, a methodology is developed to test and detect vulnerabilities in Web Services.

2. Security in Web Services

All the layers of the Open System Interconnect (OSI) stack [7] are potential targets of security attacks. At the network layer, firewalls continue to be the most popular technology for security protection from the time before the existence of Web Services. Application firewalls typically provide protection only against HTML and browser-based attacks but not against the XML message stream. However, Firewalls still guard against lower layers of communication.

The session and transport layers have Secure Sockets Layer (SSL) as part of an overall security solution recommended by companies and analysts. SSL is a good technology for protecting pipes between machines but cannot alone provide security protection for Web Services because SSL and IP filtering operate on the machine level and it does not provide application, or
user-level authentication, authorization nor provides encryption and non-repudiation at the element level. If any of the endpoints or intermediaries is compromised, then transport layer security provides no protection against attacks sent from a compromised machine.

Another concern for Web Services is about applications behind the Web Service interfaces. There can be small application packages, internally developed applications, or desktop applications. These applications can carry their own security vulnerabilities which are likely to be more exposed via Web Services interfaces.

The use of WSDL documents poses yet another serious security issue. This document is used to publicly publish the structure of Web Services in the form of XML-formatted data. This information includes the service name and location, methods name and arguments, return value and type and documentation regarding the service. This exposure accounts for the flexibility of Web Services. This makes hacking easier. Table 2 shows the growth of security incidents as reported by Community Emergency Response Team (CERT) [5].This rising vulnerability rate is of great concern to businesses. The above statistics shows the urgent needs for proper security testing of Web Services.

The major concerns in testing the security of Web Services are the lack of security testing standards and testing specifications. Since Web Services are new, not much statistics are available about the vulnerabilities and exploit patterns.

### Table 1. Security incidents

<table>
<thead>
<tr>
<th>Year</th>
<th>Incidents</th>
<th>Advisories</th>
<th>Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1989</td>
<td>132</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>252</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>406</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>1992</td>
<td>773</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>1,334</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>2,340</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>2,412</td>
<td>18</td>
<td>171</td>
</tr>
<tr>
<td>1996</td>
<td>2,573</td>
<td>27</td>
<td>345</td>
</tr>
<tr>
<td>1997</td>
<td>2,134</td>
<td>28</td>
<td>311</td>
</tr>
<tr>
<td>1998</td>
<td>3,734</td>
<td>13</td>
<td>262</td>
</tr>
<tr>
<td>1999</td>
<td>9,859</td>
<td>17</td>
<td>417</td>
</tr>
<tr>
<td>2000</td>
<td>21,756</td>
<td>22</td>
<td>774</td>
</tr>
<tr>
<td>2001</td>
<td>52,658</td>
<td>37</td>
<td>2,437</td>
</tr>
<tr>
<td>2002</td>
<td>82,094</td>
<td>37</td>
<td>4,129</td>
</tr>
<tr>
<td>2003</td>
<td>137,529</td>
<td>28</td>
<td>3,784</td>
</tr>
</tbody>
</table>

### 3. Web Applications versus Web Services

The functions, services and data of Web Services can be distributed to a number of processors without regard to hardware platform, operating system, programming language, or network topology. Web Services security cannot only rely on Web security. However, Web security cannot be ignored as it offers the underlying security for Web Services. The vulnerabilities targeting HTTP implementation can be harmful to Web Services as well and are potentially more dangerous than in the case of CGI applications; because of the nature of the business processes implemented using SOAP.

Web Services vulnerability domain can be looked upon as a subset of Web Application vulnerability domain. The analysis of the vulnerabilities exploited for the Web Application can be helpful in mitigating similar risks associated with Web Services.

Table 2 shows the Open Source Vulnerability Database (OSVDB) classification of attack types [9] and Table 3 shows the growth of vulnerabilities based on this classification [3], [7]. It can be seen that input manipulation is the most common vulnerability. Also, input manipulation, information disclosure and denial of service together constitute more than 80% of the vulnerabilities reported (Table 4). These map to the security triads of integrity, confidentiality, and dependability.

### 4. Software Vulnerability Analysis

#### 4.1. A Web Services Vulnerability Fault Model

It is a challenging task to analyze Web Services software vulnerabilities. The systems used in Web Services usually are a distributed system that is built to facilitate application to application interactions over a wide variety of hardware and software platforms and protocols. This interaction involves multiple trust boundaries and domains. Hence ensuring security and trustworthiness in this environment can be a daunting task. This can be done using a vulnerability fault model. A generic vulnerability fault model for Web Services is shown in Figure 1. It encompasses the UDDI registry, intermediate services, and the service provider host components.

#### 4.1.1 Security at UDDI registry

The UDDI registry is a critical component that allows dynamic discovery and binding in Web Services scenario. Hence it is a fundamental component of Web Services infrastructure and deserves highest level of security. Security concerns for the UDDI include information disclosure, availability and unauthorized access. Attackers can use UDDI to find information about published Web Services and use it to perpetrate the attack. A more serious form of attack is the Denial of Service attack against the UDDI server, which prevents clients from discovering the Web Services and effectively rendering them useless.
4.1.2 Security at client and intermediary services.
Web Services inputs come from a number of sources. The most obvious source is the Web Service client. Not all messages sent by the client reach the Web Service directly. These may be routed through intermediaries. Many things can go wrong while processing inputs from these sources. A message passing through an intermediary may get corrupted in unexpected ways. An attacker may take over an intermediary and cause messages to be routed to a different destination, resulting in a form of denial of service attack on the Web Service. These sources can cause a Web Service to fail and are considered untrusted.

4.1.3 Security at service host.
A service provider may have Web Services spread over multiple tiers. The Web Services code interacts with several software components, such as web servers, application servers, XML parsers, Web Services management middleware, and databases. Security may be compromised if any of these components fail to function properly.

### Table 2. Classification of attack types

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Management</td>
<td>Brute force, Default password, Cookie poisoning</td>
</tr>
<tr>
<td>Cryptographic</td>
<td>Weak encryption (implementation or algorithm), No encryption (plaintext), Sniffing</td>
</tr>
<tr>
<td>Denial of Service (DOS)</td>
<td>Saturation, Flood, Crash, Lock up, Forced reboot</td>
</tr>
<tr>
<td>Hijacking</td>
<td>Man-in-the-middle, IP spoofing, Session timeout/take-over, Session replay</td>
</tr>
<tr>
<td>Information Disclosure</td>
<td>Comments, Passwords, Fingerprinting, System information</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>DNS poisoning, Route manipulation</td>
</tr>
<tr>
<td>Input Manipulation</td>
<td>Cross-site scripting, SQL injection, File retrieval Directory traversal, Overflows</td>
</tr>
<tr>
<td>Misconfiguration</td>
<td>Default files, Debugging enabled, Directory indexing</td>
</tr>
<tr>
<td>Race Condition</td>
<td>Synchronization bugs</td>
</tr>
<tr>
<td>Other</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 3. Software vulnerabilities (by quarter) classified by attack types

<table>
<thead>
<tr>
<th>Date</th>
<th>Authentication</th>
<th>Cryptographic</th>
<th>Denial of Service</th>
<th>Hijacking</th>
<th>Information Disclosure</th>
<th>Infrastructure</th>
<th>Input Manipulation</th>
<th>Misconfiguration</th>
<th>Race Condition</th>
<th>Other</th>
<th>Unknown</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jul 1 04 - Oct 1 04</td>
<td>30 (5.6%)</td>
<td>5 (0.9%)</td>
<td>78 (14.6%)</td>
<td>10 (1.9%)</td>
<td>85 (15.9%)</td>
<td>2 (0.4%)</td>
<td>269 (50.5%)</td>
<td>32 (6.0%)</td>
<td>14 (2.6%)</td>
<td>8 (1.5%)</td>
<td>-</td>
<td>533</td>
</tr>
<tr>
<td>Apr 1 04 - Jul 1 04</td>
<td>20 (4.3%)</td>
<td>1 (0.2%)</td>
<td>47 (10.1%)</td>
<td>1 (0.2%)</td>
<td>91 (19.6%)</td>
<td>1 (0.2%)</td>
<td>266 (57.3%)</td>
<td>15 (3.2%)</td>
<td>8 (1.7%)</td>
<td>13 (2.8%)</td>
<td>1 (0.2%)</td>
<td>464</td>
</tr>
<tr>
<td>Jan 1 04 - Apr 1 04</td>
<td>27 (8.0%)</td>
<td>4 (1.2%)</td>
<td>23 (6.8%)</td>
<td>1 (0.3%)</td>
<td>50 (14.7%)</td>
<td>-</td>
<td>207 (61.0%)</td>
<td>6 (1.8%)</td>
<td>17 (5.0%)</td>
<td>3 (0.9%)</td>
<td>1 (0.3%)</td>
<td>339</td>
</tr>
<tr>
<td>Oct 1 03 - Jan 1 04</td>
<td>4 (3.1%)</td>
<td>1 (0.8%)</td>
<td>9 (7.1%)</td>
<td>1 (0.8%)</td>
<td>18 (14.1%)</td>
<td>-</td>
<td>86 (67.7%)</td>
<td>3 (2.4%)</td>
<td>2 (1.6%)</td>
<td>3 (2.4%)</td>
<td>-</td>
<td>127</td>
</tr>
<tr>
<td>Jul 1 03 - Oct 1 03</td>
<td>7 (5.0%)</td>
<td>1 (0.7%)</td>
<td>20 (14.3%)</td>
<td>-</td>
<td>20 (14.3%)</td>
<td>-</td>
<td>83 (59.3%)</td>
<td>7 (5.0%)</td>
<td>2 (1.4%)</td>
<td>-</td>
<td>-</td>
<td>140</td>
</tr>
</tbody>
</table>

### Table 4. Percentage of top three attack types

<table>
<thead>
<tr>
<th>Date</th>
<th>DOS / Information Disclosure / Input Manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jul 1 04 - Oct 1 04</td>
<td>81%</td>
</tr>
<tr>
<td>Apr 1 04 - Jul 1 04</td>
<td>87%</td>
</tr>
<tr>
<td>Jan 1 04 - Apr 1 04</td>
<td>82.5%</td>
</tr>
<tr>
<td>Oct 1 03 - Jan 1 04</td>
<td>88.9%</td>
</tr>
<tr>
<td>Jul 1 03 - Oct 1 03</td>
<td>87.9%</td>
</tr>
</tbody>
</table>
Web Services interaction with non-conspicuous sources like the operating system, file system, external libraries and other APIs should also be considered in the context of security. These sources are often considered as trusted sources by programmers. But, this trust can lead to unhandled exceptions and unexpected behavior when they fail to function as intended.

Web Services constitute complex software with many components. Trusting a component can leave the Web Service vulnerable to bugs. Hence it is important to determine the trust boundaries and understand how the components could fail. A vulnerability fault model can aid in understanding the boundaries of trust. Testing should encompass all sources that cannot be trusted. Tests should be generated by causing each component to fail.

4.2 Guidelines for Software Vulnerability Analysis in Web Services

The Fishbone chart in Figure 2 was developed by performing a root cause analysis on the vulnerabilities in Web applications disclosed between June 2003 and June 2004 [9]. Figures 3-5 provide more detailed vulnerability analysis for the top three attack types: input manipulation, denial of services, and information disclosure vulnerabilities in Web applications, because they account for more than 80% of the total vulnerabilities reported during the period (October 2003 – September 2004). Though the source data is not specific only to Web Services vulnerabilities, the analysis and result is useful to Web Services. This is because Web Services are built on top of Web application infrastructure. Web Services inherit almost all kinds of traditional Web application vulnerabilities while opening door to new type of attacks at application or data level as its technology is relying on SOAP and XML messages.

The testing guidelines are derived by first using the information from the Fishbone chart in Figure 2 to identify the common motives, modes, and targets of these attacks. Then, the vulnerability fault model is used to analyze how similar attacks may be executed and propagated in Web Services.

The fault model shows the threat profiles of Web Services and highlights the three critical areas: the registry, the client, and the Web Services. In this paper we develop the testing guidelines by focusing on the Web Services area which is the host and the application. The testing guidelines are presented as follows.

4.2.1. Check for Cross-site Scripting. SOAP and XML are standards and tools used to wrap around data for easy consumption for Web Services. SOAP provides enveloping information to deliver messages in a seamless fashion between Web Services, Web Service providers, and Web Service consumers. XML includes metadata to describe the structure of the information.

Malicious code can be embedded into the elements or CDATA of the information. CDATA is used to delineate information in the message that should not be parsed. Embedded characters or malicious code can be sent. The receiving application may display or execute the data in unintended ways. Cross-site can be used to embed commands that can tie up system resources or gain unauthorized access. Hence Web Services that display the contents of a CDATA section directly in the browser are extremely vulnerable to such attacks. Cross-site Scripting vulnerability [14] can be detected by injecting an illegal script into a SOAP message using CDATA as in the following.

```xml
<studentInfo>
  <Name><![CDATA[<script>while(true){}/</script>]]></Name>
</studentInfo>
```

If the security is compromised, the field value, which eventually is displayed in a browser, will actually run the illegal code on a browser causing an infinite loop.

4.2.2. Check for SQL Injection. If the user-supplied input is used in SQL queries without adequate validation, it may be possible for malicious users to access the data through the technique of SQL Injection. This is done by appending data and commands to an SQL query, forcing the server to send back sensitive information [4]. A Web Service can be easily compromised by sending code fragments within the envelope of Web Services. When the code fragment is unwrapped and sent to the database, special characters can cause SQL, XPATH and XQUERY statements to be executed.
SQL Injection detection can be done on Web Services by using traditional fault injection technique, a dynamic analysis process used for security software assessment [18]. Specially crafted malicious input patterns are used as input data. Responses are observed and evaluated to determine if the Web Services are vulnerable to this kind of attack.

**Figure 2:** Fishbone analysis for overall Web Services software vulnerabilities

**Figure 3:** Fishbone analysis for Input Manipulation vulnerabilities

**Figure 4:** Fishbone analysis for Denial of Services vulnerabilities
The check/testing cases applied with the technique should cover the patterns shown below.

1) Check to see if any kind of validation is performed on the user input used in database query.

Insert a single quote in the parameter field and send the SOAP request to the service. If the input is not validated or sanitized, this will yield an access denied message or a more specific one like ODBC error. Such a message is the first sign of the existence of a vulnerability that may be exploited.

2) Check for authentication bypass attacks if the Web Service uses a log-in mechanism in which usernames and passwords are used.

Passwords are generally stored in a database and are retrieved to authenticate the user based on the identification provided by the user. A query is constructed based on the input supplied by the user as shown below.

```
SELECT * FROM TABLE WHERE Username = "user-supplied-input" AND Password = "user-supplied-password"
```

If no sanity checking is done on the input before using it in the query, it may be possible to format the query to
```
SELECT * FROM TABLE WHERE TRUE
```

3) Check with wildcard. SQL uses a "%" sign to define wildcards (missing letters in the pattern) before and after the pattern. The syntax is as shown below.

```
SELECT column FROM table WHERE column LIKE pattern
```

If the Web Service uses this query, it may be possible to access other records by passing % wildcard character as the input parameter. Hence testing should be done by sending the wildcard character to see if the application accepts it.

4) Check with the query delimiter ";". Test the Web Service by appending a semicolon to the end of a parameter. The insertion of a semicolon ";" often permits the user to append a whole new SQL command to the end of the first query and to have it executed with the rights of the Web Service.

5) Check with other SQL meta-characters and their hex equivalents. Sometimes the Web Service may strip SQL meta-characters, but may fail to strip their hex equivalents. Hence it is important to test with the hex equivalents too. The following regular expression is useful in deriving test cases, including test cases for the steps mentioned above.

```
/\(\%27\)\(\%23\)\(\%22\)\#ix <TD tr>
```

The above regular expression includes testing with single quote and its hex equivalent as well as double-dash and its hex equivalent. These are SQL characters for MS SQL Server and Oracle, which denote the beginning of a comment, and everything that follows is ignored. Additionally, for MySQL, we need to check for presence of the '#' or its hex-equivalent [11].

4.2.3. Check for Script Injection. Script Injection refers to execution of operating system commands or scripts with the aid of unchecked input-parameters including XML data in SOAP message, cookies, and http request-headers. If an attacker succeeds with such a technique, it is possible to execute system commands with the same permissions of the web server.

A Web Service may use a number of system commands to call the underlying operating system to perform privileged operations on behalf of the Web Service. The OS in turn executes the command and returns the output and return-codes to the Web Service. Some examples include OS-commands for file handling like remove and copy.

If the Web Service does not perform validation and/or sanitization of the user input, system-commands can be altered in various fashions. Depending on the scripting or programming language and the operating-system it is possible through the Web Services interface to alter the system-command or alter parameters passed to system command. It may also be possible to execute additional commands and OS-typical command line tools.

The first step in performing this test is identifying the OS-specific system commands in the language used to implement the service. For example, system () and exec () in C are system commands. Testing is done by
embedding such commands in XML data passed to the Web Service and checking to see if any validation is done on the input. Tests should be also done by including command-specific meta-characters and strings in the input. Some examples are given in Table 6 [4].

Table 6. Samples of meta-characters

<table>
<thead>
<tr>
<th>Meta-character/String in Input</th>
<th>Corresponding System Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code> </code></td>
<td>Unix pipe</td>
</tr>
<tr>
<td><code>%00</code></td>
<td>Unix null</td>
</tr>
<tr>
<td><code>%20</code></td>
<td>space</td>
</tr>
<tr>
<td><code>;</code></td>
<td>command separator</td>
</tr>
<tr>
<td><code>,</code> , <code>,</code></td>
<td>Unix change directory</td>
</tr>
<tr>
<td><code>/bin/ls</code></td>
<td>Unix ls, or other commands</td>
</tr>
</tbody>
</table>

If no validation and sanity checking is performed, the Web Service is vulnerable to Script Injection. A vulnerable service is at the risk of DOS attacks since the attacker is virtually gaining system level access through OS commands.

4.2.4. Check for services traversal. For Web Services, this type of attack is analogous to file/directory traversal attacks in Web applications. The ultimate goal of the attacker resorting to this kind of an attack is to find out and gain access to Web Services that are not made public via WSDL.

The parameters are used in WSDL to convey client-specific information to the Web Service in order to execute a specific remote operation. And since instructions on how to use parameters are explicitly described within a WSDL document, malicious users can play around with different parameter options in order to retrieve unauthorized internal system information, gain unauthorized access, or bypass security checks. In addition, the information provided in a WSDL file may allow restricted users or an attacker to access other methods available. The vulnerabilities may be detected via a two step process comprising of WSDL scanning and parameter tampering:

1) WSDL scanning
   - Scan through WSDL file and directly access each service available. For example, consider a service that offers stock quoting and trading that has an advertised query method called requestStockQuote. Now if there is another unpublished transactional method called tradeStockQuote, it may be possible for a persistent hacker to use a combination of intuitive skills and brute force techniques to discover this unpublished method.

2) WSDL parameter tampering
   - Probe WSDL for parameters description and specially construct message based on the description and send to the different components of XML request processing system.

4.2.5. Check for Spoofing attacks. A message is said to be spoofed if it purports to come from one entity but actually has been created elsewhere or some other part of the message is incorrect or false. In the context of Web Services, this can be achieved easily by exploiting weaknesses in XML technologies and by exploiting the inherent vulnerabilities in message routing. Explained below are possible modes of conducting spoofing attacks on Web Services.

Routing Detour attack technique can be exploited onto intermediaries involved in Web Service being tested. Routing Detours are a form of a “Man in the Middle” attack which compromises routing information. Intermediaries can be “hijacked” to route sensitive messages to an outside location. Routing information (in the HTTP headers or in WS-Routing headers) can be modified en route.

4.2.6. Check for Buffer Overflows. Buffer overflows involve sending large amounts of data as input to the application. Usually a fixed amount of memory is used to hold the user input. When the input is too long for the memory allocated to it, the application will overwrite other instructions, causing it to abandon its normal behavior. Attackers may even be able to get arbitrary code executed using this technique. This may also allow users to execute instructions with the same privilege level as the application. This is extremely dangerous if the application is configured to run with root privileges.

A buffer overflow attack is aimed at the service endpoint and preys on vulnerabilities there, for example, not setting aside enough memory to deal with a large variety of inputs. Hence, the test is conducted by sending large inputs to the Web Service.

If the Web Service is not defensively built, it may crash causing a DOS scenario. The test should be repeated by incrementally increasing the size of input. If the Web Service physically truncates the input size or places any constraint on the input size, there is no cause of concern for buffer overflow.

4.2.7. Check for forcing error. Error messages indicate that an improper action was attempted. By trying to cause error messages, we are actually testing the robustness of the application to bad input. This may be used to uncover situations that are not handled properly [4]. These test cases can be used to see if any kind of validation is performed on input. This includes testing every part of HTTP header, cookies, SOAP header elements and SOAP body elements. If an application
fails to gracefully handle an illegal input, it may crash or allow illegal operations to be performed. Web Services use WSDL to describe the type of the input accepted by the service. Based on this information, a good set of test cases can be generated as follows.

1) Using data type different from the one expected by the Web Service
2) Entering values longer than that expected by the Web Service
3) Using boundary values including values less than expected minimum, 0, and greater than expected maximum
4) Using a different character set
5) Using a null value
6) Using white spaces and other special characters relevant to the language and platform
7) Using a format different from the one expected

4.2.8. Check for DOS attacks. Denial of Service attacks are used by attackers to prevent the service from functioning as intended. Typically it results in loss of availability [17]. These attacks are usually implemented by depriving the application of its resources. A DoS scenario is created by overwhelming the processing capabilities of the system or by installing malicious code.

It is not always possible to prevent such attacks. However, some checks/tests should be done to see if the system can handle such situations.

1) Check for buffer overflows. One of the main reasons for DOS attacks is buffer overflow. Executing tests to detect buffer overflow is the first step in detecting the possibilities of DOS attacks.
2) Perform checks to exercise the loops in the program flow. Test models like the control flow graph may be used to identify the test cases. For each loop, test cases must be executed to exercise the boundary conditions to see if an infinite loop is a possibility. In many cases, simple logical errors and bugs in software can cause infinite loops.
3) Check by issuing repetitive SOAP message requests in an attempt to overload a Web Service. A DoS attack may be caused if the Web Service does not impose and limits on the number of messages that can be sent by a single agent. Tests should be done by sending a large number of SOAP messages from a single source and monitoring the resource usage.
4) Check by generating large XML SOAP messages and sending them to the Web Service. The goal is to find if there is a limit imposed by the application on the size of XML documents. If not, the Web Service may be vulnerable to oversized payload attacks.
5) Check by sending a recursive payload. The SOAP specification says that a SOAP message must not contain a Document Type Declaration (DTD) and that on receipt of a SOAP message containing a DTD, a SOAP receiver MUST generate a fault with fault code of "Client.DTD". This specification maps to the security requirement of preventing recursive payload attacks.

4.2.9. Check for access validation. Inadequate or missing validation while allowing access to resources can lead to information leakage. Default accounts and weak passwords are commonly exploited vulnerabilities in this class. An example is the IBM WebSphere Commerce server vulnerability reported in bugtraq, which allowed the default user to access sensitive customer data. Missing validation is another major threat. A lot of attacks target resources that are not protected by proper access control mechanisms. All resources including databases, file system, kernel memory and application resources have to be protected by access control.

5. Conclusions

A methodical approach on assessing the security of Web Services with a hacker’s mindset is presented in this paper. The methodology for analyzing software vulnerability is derived by analyzing the vulnerabilities in Web Applications and extrapolating the implications to the Web Services domain and systematically incorporating tests to identify such vulnerabilities in Web Services [15]. The methodology for the software vulnerability analysis is derived by following a set of heuristics. The main heuristic is that a vulnerability already known to exist and exploited at least once will be likely exploited again. This is because it is easier to get information about the vulnerability and mode of attack [16]. There is no time involved in discovery and in devising the plan of execution. Other heuristics are:

1) No input is to be trusted
Most of the attacks target weaknesses in input validation routines, be it user input or system input. Hence testing should involve the checks and tests to detect the existence of positive validation routines and tests to exercise the validation.
2) No security mechanism is to be trusted
Mere existence of a security mechanism does not guarantee complete security. The myriad of authentication bypass attacks commonly found today are pointers to this heuristic. Hence testing should encompass tests to detect the presence of a security mechanism in place and exercising the mechanism.
3) A successful check/test in a particular configuration may fail in a different configuration.

A security mechanism may work only in a particular configuration. If the configuration of the program or the security mechanism is changed, the program may be at risk. Hence it is important to exercise the checks/tests for all possible configurations.

Checking and testing a Web Service does not guarantee the absence of malicious content. But, it is the only way to demonstrate a level of trust. The methodology developed for Web Services security testing covers the testing for the major vulnerabilities that have been reported in OSVDB. Security testing never ends as the methodology and test cases should be updated from time to time to ensure the highest security level of Web Services.

References


