Concordia University College of Alberta

Master of Information Systems Security Management (MISSM) Program

7128 Ada Boulevard, Edmonton, AB

Canada T5B 4E4

Measuring performance of two Application Servers for Java developed Web Services under heavy loads

by

GILBERT, Vincent

A research paper submitted in partial fulfillment of the requirements for the degree of

Master of Information Systems Security Management

Date: May 2009

Research advisors:

Pavol Zavarsky, Director of Research and Associate Professor, MISSM

Dale Lindskog, Associate Professor, MISSM

Measuring performance of two Application Servers for Java developed Web Services under Heavy Loads

by

GILBERT, Vincent

Research advisors:

Pavol Zavarsky, Director of Research and Associate Professor, MISSM

Dale Lindskog, Associate Professor, MISSM

Reviews Committee:

Andy Igonor, Assistant Professor, MISSM
Dale Lindskog, Assistant Professor, MISSM
Ron Ruhl, Assistant Professor, MISSM
Pavol Zavarsky, Associate Professor, MISSM

The author acknowledges the significant contributions to the work by Academic Advisors and Review Committee Members and gives the right to Concordia Univeristy College to reproduce the work for the Concordia Library, Concordia Websites and Concordia MISSM classes.

Concordia University College of Alberta

Information Systems Security Management

7128 Ada Boulevard, Edmonton, AB

Canada T5B 4E4

Measuring performance of two Application Servers for Java developed Web Services under heavy loads

by

Vincent Gilbert

vgilbert.pro@gmail.com

Research advisors:
Pavol Zavarsky, Director of Research and Associate Professor, MISSM
Dale Lindskog, Assistant Professor, MISSM

February 2008

Abstract

Businesses are increasingly migrating legacy web applications towards Web Services (WS) however there is a limited choice available for platforms which can support this technology. Furthermore these same legacy applications are increasingly the prey of DoS attacks in order to deprive businesses of their ability to operate normally. Many security mechanisms exist to protect the confidentiality and the integrity of Web Services but there is little emphasis on availability. During this research I have developed a secure java Web Service which I deployed on two separate platforms, the Oracle Business Process Execution Language (BPEL) Process Manager (PM) as well as the Sun Java System Application Server 9.1 under a Windows operating system. I then tested the performance of these afore mentioned configurations by exposing them to heavy concurrent load situations with varying types of requests. Results were concluding as both platforms were vulnerable to DDoS attacks but yet performed very differently. The Sun platform's latency was at best 5 times higher than the Oracle platform, and its throughput was in the best scenario 4 times lower than its Oracle counterpart.

1. Introduction

As businesses rely increasingly on Web Service technology to deploy critical IS assets, they proportionally expose key backend systems (Application servers, databases, LDAP, etc.) through these Web Services. Therefore Web Service security is critical and should be handled with care.

Strong and efficient mechanisms have been developed to secure Web Services data confidentiality and integrity through the emergence of standards such as WS-Security, SecureConversation or WS-Trust. However little has been done towards securing availability of these IS assets. This is even more alarming when one looks at the findings of Arber Networks inc. who surveyed 36 tier 1, tier 2 and hybrid IP network operators and found that 90% of the participants identified DDoS to be the main threat they were facing [8].

The aim of this research is to provide businesses with a comparative of performances of two main platforms supporting Web Services, faced with DDoS attacks using different types of requests.

This paper will be constructed as follows. This introductory part will be divided into two main sections, the first section will deal with the functioning of web services as well as their related security features. Companies who want to migrate from web applications to web services will often have requirements for security, therefore for this research to be both accurate and useful, it has to take into account the effects of adding security features to the performance of these platforms supporting the web services. The second section of the introductory part will provide an overview on the type of DoS which will be used for the experimentation, additional information on various types of DoS attacks can be found in this paper [7] as well as in section B.5 of this document [4]. The next part will describe the experimental methodology that I will put into place. I will then provide the results that I have gathered through several tables (figure 7, 8, 9, 10 and 12, 13, 14) and compare results from both platforms. Finally the last section will sum up my findings and my comments on the results I have found.

1.1. Web Services and the SOA paradigm

The service oriented architecture (SOA) is a paradigm where functionality is broken up into small parts, called web services, deployed at various places of a network, inside or outside of a single company. The purpose of SOA is to allow these different web services to communicate by exchanging data and thus being part of a business process no matter the underlying programming language or operating system. This architecture allows companies to design, develop and deploy reusable web services, independently, and to later assemble them together through the use of standard-based communication protocols. This allows companies to develop systems that are scalable, extendable, evolvable and therefore cost-effective.

1.1.1. SOAP

SOAP is a XML-based messaging protocol fundamentally allowing a one-way transmission between a SOAP sender and a SOAP receiver;

however SOAP is generally used to perform remote procedure calls allowing request response dialogues. SOAP is platform and programming language independent allowing it to be very interoperable; SOAP is, in the majority of cases, transmitted using HTTP as a transport protocol even though other protocols are sometimes used such as SMTP and POP. The main reason HTTP is the preferred transport protocol used for SOAP is that corporate firewalls are already setup to accept HTTP traffic whereas the usage of other transport protocols will necessitate firewalls and other network devices to be reconfigured, exposing the network to security breaches and creating lengthy discussions with the network administrators.

1.1.2. Web Services Description Language

WSDL is a language used to describe web services which is structured using the XML format. A WSDL file describes messages and ports both at an abstract and concrete level. At the abstract level the WSDL will describe messages exchanged between the requester and the provider, and port types which are abstract collections of supported operations. At the concrete layer the WSDL describes bindings which specify the transport and wire format details for the interfaces; endpoints which associate bindings with network addresses; and finally services group together endpoints that implement interfaces. [2]

Thanks to the WSDL web services can interconnect by determining how the messages they will exchange will be structured.

1.1.3. Web Services

A web service is a software system which allows to provide a functionality on behalf of its owner to a remote machine or system, no matter the underlying programming language or operating system, and this thanks to the use of SOAP as a messaging protocol. The machine or entity offering the web service is known as the provider and the machine or entity consuming the web service is known as the requester. Web services have "an interface described in a machine processable format (specifically WSDL)" [1].

The W3C describes 4 broad steps to the use of a web service. These steps describe a typical use that could be made of a web service, however in practice one could identify many other steps or a different order in which they occur. The first step is the discovery of the web service. In most cases this step is initiated by the requester. Therefore the requestor has to find or become aware of the provider, which in practice means finding out the address of the provider. This can be done either by questioning directly the provider entity if the requestor has this information or otherwise use a discovery service such as interrogating the UDDI registry. This registry is an authoritative store of information which contains information (WSDLs in particular) for many web services with the associated addresses so as to allow requesters to find the appropriate provider. One could make an analogy with a DNS server providing IP addresses to machines which are trying to contact a server on a particular domain. In addition there are other mechanisms than UDDI registries such as indexes, which are not authoritative and do not allow the provider owner to determine the data which is contained in the index as it is for UDDI registries, and finally peer-to-peer discovery.

The second step consists in the requestor agreeing with the provider to the service description and the semantics which will govern the way both parties will interact with one another. In practice this means that by reading the WSDL file the requester will understand the exact way in which the request must be formulated for it to be accepted by the provider, as well as the form in which the provider will reply to the requester.

The third step sees the requester and the provider implement the semantics on which they have agreed, this is generally more the case for the requestor which adapts itself to the provider's semantics.

The fourth step is when the requester and the provider actually exchange the SOAP messages.

1.1.4. Web Services Security

Web services suffer from the same security issues as regular web applications: buffer overflow

attacks, cross-site scripting (XSS) and distributed denial of service (DDOS). Web services also suffer from the fact that several transport layer protocols can be used such as HTTP, SMTP, or TELNET which means that firewalls have to let through many types of traffic to many different ports increasing the vulnerability of the application server. Additionally and most importantly interfaces for applications and legacy web applications were not publicly known and therefore not as easily accessible, whereas in the web service context, UDDI registries provide a listing of web services interfaces along with their WSDL file which gives a description of the interface and how to access it. This increases the visibility of these resources and therefore increases the probability with which they will suffer attacks.

In the beginning web services were secured the same way as were legacy web applications: at the transport layer using SSL/TLS or IPSEC which offered security features such as authentication, confidentiality and integrity. The problem is that nowadays web service topologies are complex and include a number of gateways, proxies, load balancers, and even other web services who act as intermediaries between the requester and the provider. SSL/TLS or IPSEC will be able to offer security between a provider and an intermediary for example, but then the intermediary will have to use SSL/TLS again between itself and the requester. In the end the requester has no other choice than to trust the intermediary to not have meddled with the data which it received from the provider. Therefore something had to be done to provide end-to-end security instead of point-to-point security.

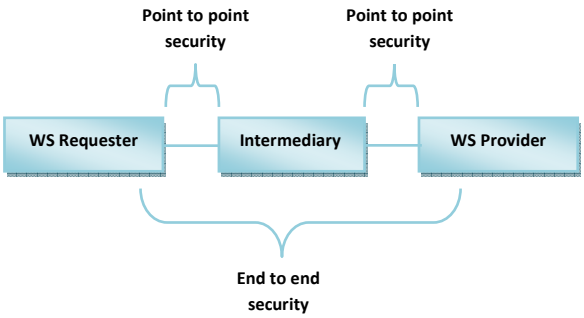

Figure 1: End-to-end compared to point-to-point security

1.1.4.1. WS-Security

SOAP as it was originally specified does not provide any security features, however this was modified with the emergence of the WS-Security standard. This standard provides several security features which are incorporated in the header of the SOAP message. These security features use XML security standards which are based on well-known cryptographic and security technologies as well as emerging XML technologies in order to provide a flexible and extensible end-to-end security solution at the message protocol layer.

The first feature provides message integrity by using XML signature. It consists of a SignedInfo element which specifies which part of the document has been signed with what algorithm, and a KeyInfo element which provides information to the recipient about the key and the eventual manipulation which has to be done with the signature before validating it.

The second feature provides message confidentiality by using XML encryption, which allows encrypting the data using various asymmetric cryptographic algorithms. The SOAP header will contain an EncryptedData element which withholds a cipher text as well as a KeyInfo element which provides information to what keying material to use to decrypt the data.

The third feature provides SOAP messaging with a Public Key Infrastructure (PKI) by using the XML Key Management Specification (XKMS) which defines a protocol to distribute and register public keys. This feature is essential to the functioning of XML Signature and XML encryption because a private key is necessary to sign or encrypt the data and a public key is necessary to verify the signature or decrypt the data. Therefore the web service provider must make sure that the public key is sent to the requester in order for this one to acknowledge or be able to read the data. [3]

WS-Security provides web services with a fourth feature: authentication. This feature supports many different security tokens such as usernames, X.509 certificates, Kerberos tickets, SAML assertions and REL tokens.

The main drawback of WS-Security is that it uses asymmetric cryptography for the encryption and the signing, which is very processor intensive. This is why the standard SecureConversation was created allowing web services to use symmetric algorithms instead of asymmetrical ones, reducing dramatically the overhead.

1.1.4.2. WS-Trust

Because of the nature of SOA and the large number of web services which can be binded together in inter-company business processes, trust relationships need to be established between remote web services. A message signed with WS-Security XML encryption but from a machine which is not trustworthy is of no use. Therefore a trust model is necessary for web services to trust the messages they are receiving, especially in an architecture where a web service is communicating with machines outside of the organization or from a different branch of the organization.

WS-Trust is one of several trust models and is based on extensions brought to the WS-Security standard. It allows to issue, renew and validate security tokens and to establish and broker trust relationships between web services. Requesters contact the provider's Security Token Service (STS) to obtain, through a challenge-response mechanism, the appropriate claim to include in the request. Claims can be either X.509 certificates or XML-based tokens such as SAML assertions. Furthermore WS-Federation allows for several WS-Trust realms to trust each other, for example in the case of web services belonging to two separate companies but which need to communicate.

1.2. Denial of Service

A Denial of Service (DoS) attack is an action or a series of actions taken by a malevolent person or group, to prevent a service offered by an information system from being accessed by its intended users. There are three main ways in which a DoS can be performed: by consuming part or all of the resources of an information system (CPU, bandwidth, disk space or memory); by destroying or altering configuration information; by physically destroying or altering network components; or by disrupting state information such as TCP states.

The most renowned type of DoS attack is the consumption of computational resources. This category can be decomposed into the consumption of network connectivity, bandwidth or computer resources.

Attacks which aim network connectivity want to prevent a victim machine from communicating with other hosts. A typical example is a SYN flood attack where an attacker creates a number of half-open connections with the victim machine it wants to perform DoS on, until this computer has used up all the structures in which it keeps track of connections. Therefore if a rightful user tries to connect, it will be unable to do so because the computer will have nowhere to save information concerning the connection. This attack is known as an asymmetric attack as it could be performed from a dial-up connection to a computer residing on a very fast network. The attacker is not consuming bandwidth but instead he is using up all the structures the computer has reserved to save connection state information.

Attacks which aim bandwidth consumption want to prevent traffic from flowing on the network where the victim machine lies. Typical examples are the use of the ICMP protocol such as in the legendary smurf attack where the attacker floods the victim's network with ICMP echo requests with a spoofed source address corresponding to the victim machine in order for all the other machines on that network to respond with ICMP echo responses. The network the victim lies on will be jammed with ICMP echo traffic and the victim machine will be unable to respond to any legitimate requests. Another example is a peer-to-peer attack where attackers trick peer-to-peer users into making continual requests to the intended victim creating an enormous load on it. This can result in the victim's machine crashing or taking so long to respond to legitimate requests that by the time it does respond the connection has timed out. These types are especially effective if the attacker's bandwidth is greater than the victim's bandwidth.

Attacks which aim computer resources want to prevent the victim machine from being able to operate. Attackers might try to consume the victim machine's entire disk space. A typical example is the use of a worm running on the victim machine duplicating itself until the disk space is full. Another example would be a buffer overflow attack where instead of trying to run malicious code on the victim machine, the attacker only wants to fill up the victim's memory thus slowing down or crashing the victim machine altogether. Many other variants of this type of attack exist.

1.2.1. Distributed Denial of service

A Distributed Denial of Service (DDoS) is a DoS attack performed by several machines at the same time against a same victim machine. DDoS attacks are always types of attack aiming to consume the computational resources of a victim, and in most cases consuming the victim's bandwidth. As explained earlier these types of attacks are effective if the attacker's bandwidth is greater than the victim's. Therefore if there are several machines performing DoS at the same time on the same machine there are more chances for the attack to succeed. A good example of such an attack is the MyDoom worm which spread rapidly to millions of machines worldwide. The infected machines were set to launch a multitude of requests to a particular website on the 1st of February 2004, thus denying access to legitimate users.

1.2.2. Availability for WS

In the context of web services the NIST defines Quality of Service (QoS) as the assurance that a web service is consistently operating at the expected level of performance; reliability as the assurance the web service operates correctly and as expected in the presence of unexpected faults; and availability the assurance the web service operates correctly and as expected in the presence of intentional faults and that if it were to fail it would do it in a safe state.

The NIST argue that availability, Quality of Service (QoS) and reliability are related to one another as availability ensures that QoS and reliability are maintained even if there is an attempt to compromise the web service's operations, such as a DoS attack.

Furthermore it enumerates three objectives the web service should do to achieve availability. The first is to recognize the attack patterns of a DoS, the second is to shut down safely if failure is inevitable and avoid the DoS from spreading, and thirdly to recover and resume secure operations as soon as possible after a DoS attack.

This third point will be of particular interest in this research as one of the criteria we will observe will be the time the application servers will need, to recover completely from the heavy load situation.

2. Experiments

An application server will be exposed to heavy loads of requests and a number of its characteristics will be monitored to assess its behaviour. In this section I will explain all the aspects of my experiments. I begin by describing the characteristics of the platforms I have chosen to study as well as the reason for this choice. I then continue describing the characteristics of the web service I will be using for this test. The last section describes the exact experimental methodology I will be using as well as the tools and the types of requests I will be performing.

2.1. Platforms

I have chosen to test the performance of two application servers which support Java developed web services: Oracle's BPEL Process Manager 10.1.3.1.0 and Sun's Java System Application Server 9.1. Both Oracle and Sun are major actors in the application server world and have been working on the SOA paradigm and on web services since the introduction of this technology. I could have also chosen to test IBM's WebSphere application Server which is also another major actor in this field; however the trial version offered by IBM seemed to be very different from its commercial counterpart.

2.1.1. Oracle BPEL Process Manager 10.1.3.1.0 for OC4J

This application server supports Java 2EE, XML, WSIF, WSIL and WSDL standards. Furthermore it contains the Oracle Web Services Manager which allows deploying, publishing, managing, and monitoring web services. In particular Oracle's WSM allows viewing details of execution such as success, failure, failure due to lack of authentication, authorization, latency etc. It also provides graphs of these parameters over time, such as graphs of variance of latency.
These features are important as they will help me assess how the application server is behaving during the experiments.

2.1.2. Sun Java System Application Server 9.1

This application server is entirely based on the Glassfish V2 application server, which is an open source project released under an OSI approved license (CDDL). It implements Java EE 5 technologies and supports JAX-WS 2.0 and JAX-B 2.0. This server offers a BPEL engine much like Oracle's application server.

Most importantly the administrator's interface of this application server proposes several tools to monitor the performance of the application server much like Oracle's application server.

2.2. WS Deployed

During my experiments I will be testing the behaviour of these two application servers therefore I have to make sure that all the other parameters are identical for the experiments to be reliable. This is why I will be deploying an identical web service to them. This web service is very simple and is not at all greedy of resources. It consists in summing two integers that it will have received from the requester and sending back the result of this addition. By choosing to add two integers I will be able also to test how the application servers react when they will receive a request with a type mismatch and that the sum will be done with something different than integers, as this might be a strategy chosen by an ill-intentioned person to trick or slow down these servers.

2.2.1. Security features implemented

For my experiments to be useful and interesting for companies wishing to migrate their web applications to web service technologies I have implemented security features on the web service which will be tested. Therefore any impact that implementation of security features have on the performance of application servers will be taken into account in the results of the experimentation.

Through the use of WS-Security I have implemented authentication using a username and password pair as a security token for the requester to authenticate to the provider. I have decided not to implement encryption or digital signing because of the very significant overhead that they imply due to the use of asymmetric encryption. This would significantly change the results and not be accurate as this overhead can be easily mitigated using the SecureConversation standard which uses symmetric encryption.

The result is a web service which is secured and resembles to a web service a company might decide to deploy.

2.2.2. WSDL's

Here are the WSDLs for both web services:

```
<definitions
     name="SimpleAddition"

targetNamespace="http://addition.mon.org/"

xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://addition.mon.org/"

xmlns:soap12="http://schemas.xmlsoap.org/ws
dl/soap12/"

xmlns:mime="http://schemas.xmlsoap.org/wsdl
/mime/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema
"

xmlns:soap="http://schemas.xmlsoap.org/wsdl
/soap/"
    >
    <types>
        <schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://addition.mon.org/"
             elementFormDefault="qualified"
xmlns:tns="http://addition.mon.org/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl
/"

xmlns:xsi="http://www.w3.org/2001/XMLSchema
-instance" xmlns:soap11-
enc="http://schemas.xmlsoap.org/soap/encodi
ng/">
            <element name="add"
type="tns:add"/>
            <complexType name="add">
                <sequence>
                    <element name="i"
type="int"/>
                    <element name="j"
type="int"/>
                </sequence>
            </complexType>
            <element name="addResponse"
type="tns:addResponse"/>
            <complexType
name="addResponse">
                <sequence>
                    <element name="return"
type="int"/>
                </sequence>
            </complexType>
        </schema>
    </types>
    <message name="SimpleAddition_add">
        <part name="parameters"
element="tns:add"/>
    </message>
    <message
name="SimpleAddition_addResponse">
        <part name="parameters"
element="tns:addResponse"/>
    </message>
    <portType name="SimpleAddition">
        <operation name="add">
            <input message="tns:
SimpleAddition_add"/>
            <output message="tns:
SimpleAddition_addResponse"/>
        </operation>
    </portType>
    <binding name="SimpleAdditionHttp"
type="tns: SimpleAddition">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/
http"/>
        <operation name="add">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="SimpleAddition">
        <port name="SimpleAdditionPort"
binding="tns: SimpleAdditionHttp">
            <soap:address
location="http://10.0.0.2:8888/WSResProj-O-
SimpleAddition-context-
root/SimpleAdditionHttpPort"/>
        </port>
    </service>
</definitions>
```

Figure 2: WSDL of web service deployed to Oracle Application Server

```
<definitions
 xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:wsdl="http://schemas.xmlsoap.org/wsdl
/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema
"

xmlns:soap="http://schemas.xmlsoap.org/wsdl
/soap/" name="SimpleAdditionService"
targetNamespace="http://addition.mon.org/"
xmlns:tns="http://addition.mon.org/"
 >
    <message name="add"/>
    <message name="addResponse"/>
    <portType name="SimpleAddition">
        <operation name="add">
            <input message="tns:add"/>
            <output
message="tns:addResponse"/>
        </operation>
    </portType>
    <binding
name="SimpleAdditionPortBinding"
type="tns:SimpleAddition">
        <operation name="add">
            <input/>
            <output/>
        </operation>
    </binding>
    <service name="SimpleAdditionService">
        <port name="SimpleAdditionPort"
binding="tns:SimpleAdditionPortBinding"/>
    </service>
</definitions>
```

Figure 3: WSDL of web service deployed to Sun Application Server

The differences in the WSDL can be explained mainly by the way both IDE's (Jdeveloper for

Oracle and Netbeans for Sun) compile the same instructions and the detail they put in the WSDL document. However both web services perform the same operations and both WSDL's are very similar to one another.

2.3. Experimental Methodology

This section will provide you with the details of how the experiments were conducted as well as the methodology I used.

2.3.1. Network Setup

I will be using three machines throughout my experimentation. One machine will host the application servers, and the two other ones will generate the heavy loads of requests. All three machines will be on an isolated network so as to make sure that no other traffic can come on the network and affect the experimentation. By using two machines to generate requests I will be able to create a heavier load than by using one, and therefore have a bigger impact on the application server.

The machine running the application server has a 3.2 GHz AMD Athlon 64 bit processor, 2.0 GB of Ram and a 100 Mbps Ethernet card. One of the attacker machines creating the heavy load has a 1.73 GHz Intel Pentium M processor, and the second one has a 3.0 GHz AMD Athlon 64 bit processor; both have 512 MB of RAM and a 100 Mbps network card, which is largely sufficient to run Jmeter and flood the application server. All three machines are connected to one another through a 100 Mbps switch.

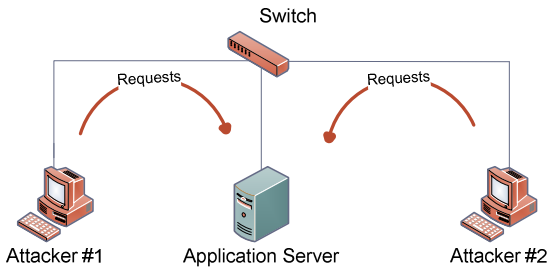

Figure 4: Network Topology

2.3.2. Jmeter

Jmeter is a tool to perform load testing of functional behaviour as well as to measure performance. It can be used to simulate heavy load on a network, on a server or on a particular object to analyze its behaviour or its performances under different types of load. It is part of the Apache Jakarta Project which is open source.

Furthermore Jmeter can test performance of HTTP, HTTPS, FTP, JDBC, etc as well as web services.

Finally Jmeter allows you to make a graphical analysis of performance by incorporating various data in a same graph.

To use Jmeter one has to create a test plan which is made of different elements such as controllers, listeners, pre/post-processors, assertions, timers, etc which allow us to completely customize any kind of test that we may decide to do. The test plan for my experiments is the following:

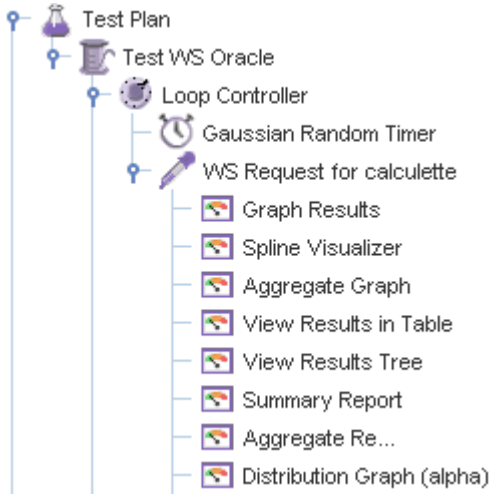

Figure 5: Test Plan for experiments

This test plan creates 500 threads which each make 10 requests the total of requests summing up to 5000. There is no ramp up time meaning all of the threads start making their requests at the same time; however I inserted a Gaussian random timer which allows the client requests to have a more chaotic distribution so as to hit the server at random intervals.

I setup different kinds of listeners to collect various data such as a graphical representation of the throughput, its average and its median, or a spline visualiser which provides a view of all the sample times allowing us to see if there is a saturation phenomenon of the server after a certain number of requests.

Because I will have two machines running Jmeter I will be able to have more precise results as well as compare them in order to detect an eventual anomaly in the experimentation not due to the application server itself (an example would be if one of the computer's memory filled up and was not able to receive to operate correctly thus altering the results of the experimentation).

2.3.3. Types of DoS performed

In my experimentation I will be attempting a DoS by consuming the bandwidth of the tested application servers. This will be achieved by creating a very large number of requests to the application servers from numerous different threads. Because the attacking machine will be on the same local subnet as the application servers, my attacks will be comparable to a DDoS, an attack performed by many machines, as the number of requests achieved per second will be far greater than what would be possible if the two attackers where on a distant network separated from the victim by many routers and a long distance.

2.3.4. Types of requests

Each application server will be tested with 3 types of requests: a normal authenticated request which asks to sum up two integers, a request with wrong authentication credentials which asks to sum up two integers, and an authenticated request which asks to sum up an integer with a character.

The second and third type of requests will produce particularly interesting results as we will see how the respective application servers react to events such as an unauthenticated request or a type mismatch and whether this hinders their performance. If this is the case it could mean that an attacker could profit from this to perform a DoS on the application servers more easily.

Here is an example of the SOAP request which is sent to one of the application servers:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap
/envelope/"    xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
       <soap:Header>
              <wsse:Security
xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
xmlns="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
xmlns:env="http://schemas.xmlsoap.org/soap/
envelope/" soap:mustUnderstand="1">
              <wsse:UsernameToken
xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
xmlns="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd">

       <wsse:Username>vincent</wsse:Usernam
e>
              <wsse:Password
Type="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-
username-token-profile-
1.0#PasswordText">pass1</wsse:Password>

       </wsse:UsernameToken></wsse:Security
>
       </soap:Header>
    <soap:Body
xmlns:ns1="http://addition.mon.org/">
        <ns1:add>
            <ns1:i>45</ns1:i>
            <ns1:j>11</ns1:j>
        </ns1:add>
    </soap:Body>
</soap:Envelope>
```

Figure 6: SOAP request

The SOAP message requests the web service to sum up two integers 45 and 11 and a simple plaintext authentication with username and password is sent in the SOAP header.

2.3.5. Parameters monitored

During my experiments I will be monitoring various parameters to assess the behaviour of these application servers. These parameters are: throughput (requests per second), average, median and maximum request latencies, percentage of requests dropped, and the length of time of the experiment. I will also look at the time

the system takes to return to a normal state once the requests have finished been all treated. I will not use the monitoring tools provided by the platforms as under heavy load conditions these have a lot of trouble providing reliable data. Instead I will use the monitoring tools provided by Jmeter and compare the results from both attackers.

From these parameters I will be able to obtain several graphs and tables indicating the performance of the application servers.

3. Results

The first step in my experimentation is to test both platforms with a small number of requests in order to establish a baseline on how the platforms perform in a normal situation. I use a test plan with 10 threads creating 2 requests each with no ramp up time. The type of request used is the first type, namely the normal authenticated requests. Here are the results for this baseline test:

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Oracle platform	14.1	402	406
Sun platform	21.7	254	266

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Oracle platform	500	0	~2
Sun platform	391	0	~1

Figure 7: Baseline test results

We can see that both platforms perform similarly even though the Sun platform offers an average latency 37% lower than the Oracle platform as well as a 50% better throughput.

Throughout this section I will be discussing the percentage of errors, which should be understood as the percentage of dropped requests.

3.1. Oracle Platform

As the results given in the following section indicate, the Oracle platform performs in a linear fashion. Requests from either attacker are dealt with in the same way, and the fact that the median latency is close to the average latency suggests that as a whole the majority of requests had approaching latencies.

3.1.1. Results for flooding with normal requests

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	18.6	24 599	19 203
Attacker # 2	18.2	25 181	19 613

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	81 156	49.3	216
Attacker # 2	83 843	34.9	226

Figure 8: Results for type 1 requests on Oracle platform

The results of the first experimentation show us that the average and median latency are very high. They average latency is 61 times what it was in the baseline test. Furthermore the percentage of errors (i.e. the percentage of requests not being answered) is between 35% and 50%. A legitimate user trying to access the service during this period of time would have between one out of two and one out of three chances of never getting the response to his request; and if he were to receive a response the user would in average get it 25 000 ms after having sent the request, which would could have resulted in a connection timeout. All

these elements demonstrate that by creating a heavy load situation using normal authenticated requests we were able to create a partial denial of service on the application server.

3.1.2. Results for flooding with badly authenticated requests

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	45.8	12 284	13 703
Attacker # 2	42.8	13 138	14 618

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	28 078	35.1	104
Attacker # 2	25 219	31.8	101

Figure 9: Results for type 2 requests on Oracle platform

We can see from these results that the average latency is 31 times what it was in the baseline test. We also find a percentage of errors between 32% and 35%. Compared to the previous experimentation we can see the platform reacts better to a heavy load of unauthenticated requests, probably because as soon as the request are identified as not properly authenticated the platform does not need to process the request and instead just answers with an error message. It is still resource consuming as the platform has had to demarshal the content, compare the security tokens with the stored encrypted value, create a response, marshal it in a SOAP message and send it on the wire.

However it is a good point that an attacker could not perform a DoS as easily if he does not have the adequate security tokens to perform a request to the web service. It complicates an attacker's task.

3.1.3. Results for flooding with requests which create a type mismatch error

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	23.9	15 293	7547
Attacker # 2	24.3	14 938	8 022

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	81 875	54.3	190
Attacker # 2	81 344	60.3	175

Figure 10: Results for type 3 requests on Oracle platform

These results are different from both previous experiments. The median latency is lower than in both previous experiments but is still around 19 times then during the baseline test. The average latency is nearly the double of the median, which indicates that a small number of requests had extremely high latencies thus creating a large difference between the average and the median. However the most important point is that percentage of errors reached 60% meaning that the platform was simply unable to respond to more than one out of two requests. The fact that these requests contained a type mismatch between what the web service was expecting and what it received, seem to indicate that the platform had problems dealing with this situation and that it was often discarding the requests altogether.

Another phenomenon which happened during all of the three experiments is that the usage of the CPU shot up to 100% during the attack, and only decreased 10 to 12 minutes after it ended. This seems to indicate that the platform has difficulty

with achieving the third objective of availability as described by the NIST [4].

In conclusion, the experiments showed that the Oracle platform was subject to DoS attacks especially with normal authenticated requests and requests containing a type mismatch. It is a very good point that requests with erroneous security tokens hindered less the platform as it prevents attackers from being able to perform a DoS if they do not have the right security tokens.

3.2. Sun platform

As we are about to see in this section, the Sun platform behaves very differently than its Oracle counterpart. It gives priority to the host it received the request from first, neglecting the second hosts requests. This has an enormous influence on the results, as the experiment can be divided into two phases. During the first phase attacker #1 receives 100% of its request back with latencies averaging 110 000ms while attacker #2 receives 10% of its requests back with an average latency of 120 000ms. Once the first attacker has received all of its requests back, the Sun platform concentrates its efforts to answer the remaining requests of attacker #2 which it does with 0% of error with average latency of around 50 000ms.

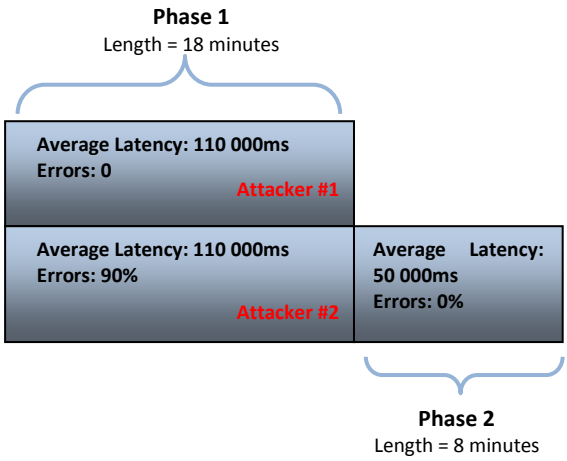

Figure 11: Behaviour of Sun platform to heavy load situations

Therefore this means that the results shown in the graphs will be an average of these two phases, and would be different if attacker #1 continuously sent requests. However my experimental methodology has to be the same to compare these two platforms, so that the results can be considered as valid.

This explains why the second attacker has a great difference between its average and median latency.

3.2.1. Results for flooding with normal requests

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	3.8	114 844	121 562
Attacker # 2	2.9	160 716	58 641

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	123 469	0	1084
Attacker # 2	482 484	45	1582

Figure 12: Results for type 1 requests on Sun platform

The results that are above are very alarming. Attacker # 1 having started the requests a few hundred milliseconds before Attacker # 2, it is given precedence for all of the requests. The Sun platform strives to answer to all of the requests of the first attacker, forgetting partially the second attacker's requests. However in doing so the platform provides a very poor service to the first attacker responding in an average time of 114 seconds, just under two minutes, which means that the connection will have possibly timed out and that the requests will not be received by the requester; the platform provides an ever poorer service to the second attacker, responding to only 55% of requests and with an average response time of 160 seconds, two minutes and a half.

The throughput obtained by both attackers averages around 3.5 requests per second which is

5 times less than the Oracle platform did under the same conditions.

These results clearly show that I was able to perform a very effective DoS on the Sun platform as in average requests are answered in two minutes, which is unacceptable for a component of a business information system.

3.2.2. Results for flooding with badly authenticated requests

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	3.9	112 965	121 169
Attacker # 2	2.9	162 346	55 178

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	122 804	0	1072
Attacker # 2	484 698	48.4	1516

Figure 13: Results for type 2 requests on Sun platform

These results are very similar to the previous results, which contrasts with the Oracle platform. The type mismatch provoked by the content of this type of request does not seem to aggravate or better the performance of the Sun platform. This is possibly due to the fact that the influence of the type of request is negligible compared to the influence the heavy load situation is having on the platform's performance.

The throughput obtained is very close to what was found in the previous experiment. This seems to confirm the results as the behaviour of the platform seems to be consistent.

3.2.3. Results for flooding with requests which create a type mismatch error

	Throughput (req/sec)	Average Latency (ms)	Median Latency (ms)
Attacker # 1	3.7	116 285	121 515
Attacker # 2	2.9	165 171	54 500

	Maximum Latency (ms)	% of dropped requests	Length of Experiment (s)
Attacker # 1	122 313	0	1096
Attacker # 2	487 797	50.1	1544

Figure 14: Results for type 3 requests on Sun platform

The results seem to show that the type of requests does not alter the way in which the Sun platform behaves, one attacker sees all its requests being answered while the other attacker gets a response for one out of two requests, and most importantly the responses are received very late to the point that the receiver will probably discard them.

4. Analysis of Results

Results above prove how both platforms hosting a Web Service are vulnerable to DDoS attacks, however both platforms behaved differently.

The three types of request had varying impacts on the Oracle platform. The first experiment with type 1 requests (normal authenticated request supplying correct arguments) provoked the highest latencies whereas type 3 requests (normal authenticated request supplying wrong arguments, an integer and a character) lead to the highest percentage of dropped requests. Finally attacking with type 2 requests (badly authenticated request supplying correct arguments) was the least efficient, causing smaller latencies and smaller percentage of dropped requests. This is very

positive as attackers require correct credentials to efficiently attack Web Services hosted on an Oracle platform.

On the other hand the Sun platform performed identically for all three types of requests. This is problematic as any type of request will allow to perform a DDoS attack on a Sun platform hosting a Web Service, including one with erroneous credentials.

Overall in terms of request latencies, the Oracle platform behaved better than the Sun platform as for type 1 requests Oracle platform saw latencies 5 times lower than for the Sun platform, for type 2 requests 11 times lower, and for type 3 requests, 9 times lower.

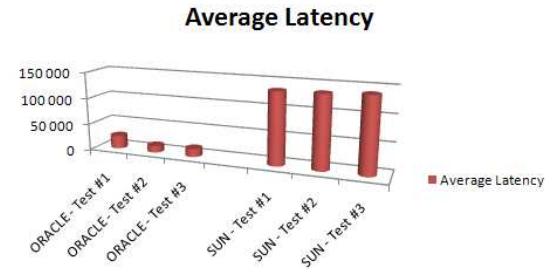

Figure 15: Graph of average latency per request and platform type

These latencies impact on the time needed to handle the 10.000 requests sent by both attackers. For type 1 requests, the Oracle platform needs 7 times less time than its Sun counterpart to handle all the requests, for type 2 requests over 14 times less, and for type 3 requests 8 times less.

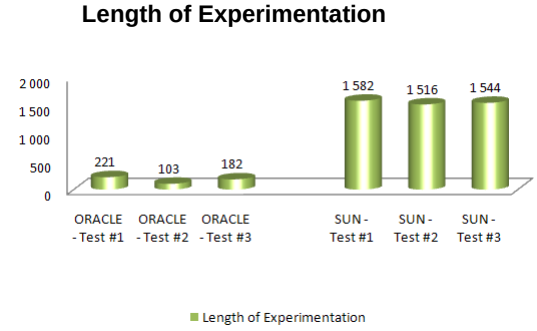

Figure 16: Graph of average length of experimentation per request and platform type

More importantly both platforms behave differently in terms of handling requests. The Oracle platform's behaviour is steady and linear. Requests from both attackers are treated in the same way. Throughout each experiment latencies and percentage of dropped requests are quite homogeneous.

The Sun platform on the other hand seems to favour request coming from one host more than the other, which impacts on results for both attackers. During a first phase, the Sun platform drops 0% of attacker #1 requests and drops over 90% of attacker #2 requests. Once all of attacker #1 requests have been answered, the Sun platform focuses on the second attacker and drops 0% of its remaining requests. This behaviour is dangerous as if an attacker were to continuously flood the Sun platform with requests 90% of any other request coming from legitimate users trying to access the Web Service would be dropped. Furthermore this behaviour is not efficient at all, as requests coming from either attacker are answered so late that chances are normal hosts would treat the connection as having timed out.

There is one field in which the Sun platform behaves better than the Oracle one, it is the percentage of dropped requests. The Sun platform had approximately 25% of dropped requests for all three types of requests, whereas the Oracle platform dropped around 40% of requests for type 1, 33% for type 2, and 57% for type 3. However as explained above, this is due to the way the Sun platform behaves in two phases, and the figures would be different if one were to experiment with the first attacker continuously sending requests.

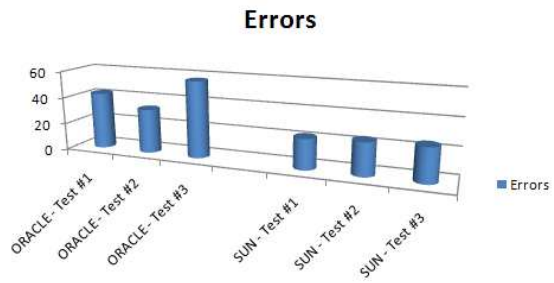

Figure 17: Graph of average errors per request and platform type

There is one other point on which the Oracle platform behaves worse than its Sun equivalent, this is after the 10.000 requests have been handled the machine which hosts the Oracle platform has its CPU usage stuck at 100% between 10 and 12 minutes; this does not happen for the Sun platform which immediately ceases to use all computational resources.

The Oracle platform behaved differently for the three types of requests which were experimented. The least efficient attack was the one performed with the erroneous security token which means that attackers would require obtaining the correct security tokens to effectively attack the platform. The attack which obtained the highest latencies was the one performed with the normal authenticated request, while the attack which obtained the most errors or dropped requests was the attack performed with the type mismatch requests.

5. Conclusion

The aim of this paper was to measure the performances of two application servers hosting a Web Service exposed to a DDoS attack. We first examined particularities of Web Services and their security mechanisms, in particular how the use of SOAP allows providing an end-to-end security mechanism at the message layer. We then looked at how a particular type of DDoS attack functions, DDoS through packet flooding. We then detailed the experimental methodology used to appropriately test the two application servers hosting the Web Service. This experimental methodology defined three types of requests we would test the Web Service against: a normal authenticated request providing two integers, a badly authenticated request providing two integers, and a normal authenticated request providing a character and an integer. The aim was to see if differences in the type of request with which the Web Service would be flooded would alter the way in which the application servers performed.

The results section above showed just how vulnerable both application servers hosting the Web Service could be. However they both performed differently.

The Oracle platform performed the best for the following reasons: even though around 50% of requests were dropped, the remaining 50% were answered in average in between 12 and 25 seconds depending on the type of requests, whereas the Sun platform dropped only 25% of requests but requests were answered in average in 137 seconds at least 5 times higher.

Secondly the Oracle platform was less vulnerable to the attack when the type of request used did not have correct authentication, which makes life harder for attackers who have to have correct credentials to successfully attack. This was not the case for the Sun platform which behaved exactly the same for all three types of requests, facilitating the task of an attacker.

Thirdly the total time needed for the Oracle platform to handle the 10.000 request was 8 times lower to the time needed for the Sun platform (between one minute and a half and three minutes for the Oracle platform, and over 25 minutes for the Sun platform).

Both platforms hosting the same Web Service were to different extents vulnerable to DDoS attacks but the Sun platform behaved much worse than its Oracle counterpart. Businesses deploying Web Services on any of these two platforms should carefully plan defence mechanisms to mitigate these serious problems.

6. Future Work

This research is only the first step towards testing these two platforms which support Web Service technology, when faced with DDoS attacks.

One limitation of my experiments was that we did not expect the Sun platform to behave in a two phase fashion (cf. figure 11) therefore it would be interesting to modify the experimental methodology in order to have one attacker continuously flood the application servers and test after various intervals the throughput which can be obtained. We believe that in this scenario the Sun platform's performances would be even poorer as the continuous flooding of the first attacker would mobilize all the platform's resources to answer the first attacker, neglecting any other hosts requests.

It would also be interesting to test both platforms against several other types of DoS attacks than the one we chose, such as sending oversized payloads or XML injection and see what DoS attack is most efficient.

More importantly future research should focus on ways to mitigate these problems. Already some solutions specific to Web Services exists such as stateful Web Service firewalls, and testing them against these types of attacks could help businesses identify the security mechanisms they should put in place to protect their asset.

7. Acknowledgments

I would like to thank my professors Pavol Zavarsky, Dale Lindskog and Andy Igonor for their kind support and their precious help.

8. References

[1] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D., (2004) Web Services Architecture

[2] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Frystyk Nielsen, H., Thatte, S., Winer, D., (2000) Simple Object Access Protocol (SOAP) 1.1

[3] Chinnici, R., Gudgin, M., Moreau, J. J., Schlimmer, J., Weerawarana, S., (2003) Web Services Description Language (WSDL) Version 2.0

[4] Shingal, A., Winograd, T., (2006) Guide To Secure Web Services (Draft) - *NIST*

[5] Gruschka, N., Jensen, M., Luttenberger, N., (2007) A Stateful Web Service Firewall for BPEL

[6] Vieira, M., Laranjeiro, N., (2007) Comparing Web Services Performance and Recovery in the Presence of Faults

[7] Mirkovic, J., Martin, J., Reiher, P., (2001) A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms

[8] Arbor Networks. Worldwide ISP Security Report (September 2005)

[9] Topley, K., (2003) Java Web Services in a Nutshell - *O'Reilly*

[10] Mulyar, N., (2005) Pattern-based Evaluation of Oracle-BPEL (v.10.1.2)

[11] Razmov, V., (2000) Denial of service attacks and how to defend against them

[12] CERT Coordination Center, Denial of Service Attacks
http://www.cert.org/tech_tips/denial_of_service.html

[13] CERT Coordination Center, (2001) Trends in Denial of Service Attack Technology

[14] Hussain, A., Heidemann, J., Papadopoulos, C., (2003) A framework for classifying denial of service attacks

[15] Mirkovic, J., Prier, G., Reiher, P., (2002) Attacking DDoS at the source

[16] Xu, J., Lee, W., (2003) Sustaining Availability of Web Services under Distributed Denial of Service Attacks