Future Generation Computer Systems 37 (2014) 37–45

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

DDoS defense system for web services in a cloud environment

Thomas Vissers [a,*], Thamarai Selvi Somasundaram [b], Luc Pieters [a], Kannan Govindarajan [b], Peter Hellinckx [a]

[a] University of Antwerp, Faculty of Applied Engineering: Electronics-ICT, Paardenmarkt 92, B-2000 Antwerpen, Belgium

[b] Anna University, Madras Institute of Technology Campus, Department of Computer Technology, Chromepet, Chennai-600 044, Tamil Nadu, India

HIGHLIGHTS

- Reported DoS vulnerabilities in web services are analyzed and confirmed.
- The impact of exploiting these application-layer vulnerabilities is devastating.
- An adaptive HTTP and XML inspecting defense system with minimal overhead is proposed.

ARTICLE INFO

Article history:
Received 13 June 2013
Received in revised form
29 January 2014
Accepted 8 March 2014
Available online 22 March 2014

Keywords:
Denial-of-service
Cloud computing
Mitigation
Web services
SOAP
XML
HTTP

ABSTRACT

Recently, a new kind of vulnerability has surfaced: application layer Denial-of-Service (DoS) attacks targeting web services. These attacks aim at consuming resources by sending Simple Object Access Protocol (SOAP) requests that contain malicious XML content. These requests cannot be detected on the network or transportation (TCP/IP) layer, as they appear as legitimate packets. Until now, there is no web service security specification that addresses this problem. Moreover, the current WS-Security standard induces crucial additional vulnerabilities threatening the availability of certain web service implementations. First, this paper introduces an attack-generating tool to test and confirm previously reported vulnerabilities. The results indicate that the attacks have a devastating impact on the web service availability, even whilst utilizing an absolute minimum of attack resources. Since these highly effective attacks can be mounted with relative ease, it is clear that defending against them is essential, looking at the growth of cloud and web services. Second, this paper proposes an intelligent, fast and adaptive system for detecting against XML and HTTP application layer attacks. The intelligent system works by extracting several features and using them to construct a model for typical requests. Finally, outlier detection can be used to detect malicious requests. Furthermore, the intelligent defense system is capable of detecting spoofing and regular flooding attacks. The system is designed to be inserted in a cloud environment where it can transparently protect the cloud broker and even cloud providers. For testing its effectiveness, the defense system was deployed to protect web services running on WSO2 with Axis2: the defacto standard for open source web service deployment. The proposed defense system demonstrates its capability to effectively filter out the malicious requests, whilst generating a minimal amount of overhead for the total response time.

1. Introduction

During a denial-of-service (DoS) attack, the attacker tries to overload the victim's resources, resulting in a reduced or denied service for legitimate users who are trying to access the victim's services. In a distributed case, the attacker recruits zombies[1] by infecting numerous machines across the Internet, creating a botnet. These distributed botnets can be used to drastically multiply the strength of the attack. Moreover, it makes it difficult to filter out malicious sources and hides the true identity of the attacker orchestrating the assault. These distributed denial-of-service (DDoS) attacks are one of the most devastating and realistic

* Corresponding author. Tel.: +32 486884069.
E-mail addresses: thomasvissers@gmail.com, thomas.vissers@student.ua.ac.be (T. Vissers).

[1] Systems under control of the attacker.

http://dx.doi.org/10.1016/j.future.2014.03.003
0167-739X/

cyber threats today, as they can effectively ruin the service, profits and reputation of any organization operating through the Internet. Furthermore, specific knowledge of the victim's infrastructure is hardly necessary. Botnets can therefore easily be re-targeted at a new victim with a minimum of effort.

With the rise of cloud computing and web services, even more dependency is put on the security and availability of resources accessible over the Internet [1]. It is crucial that services remain operational, making them an attractive target for attackers. Additionally, Economic Denial-of-Sustainability (EDOS) introduces an additional motive: these are attacks which aim at consuming resources to drive up the cost of cloud computing [2].

Recently, a new kind of vulnerability has surfaced: application layer denial-of-service (DoS) attacks targeting web services. These attacks aim at consuming resources by sending SOAP requests that contain malicious content. Generally, they target the possible heavy resource demands induced by a request to a web service. The malicious requests cannot be detected on the TCP/IP layer, as they appear as legitimate packets.

The current security specifications for web services, such as WS-Security, do not consider the problem of availability. They only address confidentiality, integrity and authorization. Moreover, these specifications themselves introduce additional weaknesses for application layer DoS attacks. It is clear that a new kind of defense system has to be designed to counter this type of attack.

As described in [3] we can distinguish between two major DoS threats against web services on the application layer: HTTP and XML attacks.

- XML attacks target web services that communicate through XML documents. SOAP mostly uses XML for passing data between the client and server. Attackers construct malformed XML requests and send these to the web service. Even a single malformed request might be extremely resource intensive to process. Hence, the attacker can cause considerable harm with a minimum amount of resources.
- An HTTP attack is very rudimentary: the attacker floods a web service with non-specific HTTP requests. As the web service tries to process all requests and the particular service requires heavy use of resources, a denial-of-service is easily achieved.

This paper proposes a system for defending against these two types of application layer threats. However, it must be made clear that the proposed defense system is specific for threats involved with web service deployment. It does not replace the lower-layer DDoS defense systems that target network and transportation attacks. Therefore, for complete protection, it is recommended that such a defense mechanism precedes the proposed system.

2. Related work

Jensen et al. [4] conducted a thorough research on the vulnerabilities in the current SOAP specifications and frameworks. Several of these attacks target the availability of the web service, such as coercive parsing, oversize payload, oversize cryptography and attack obfuscation. Furthermore, the paper provides several possible counter measures. One of the most prominent suggestions is the use of a stream-based XML parser such as SAX [5]. Originally, Document Object Model (DOM) parsers were used to process XML documents. These parsers load the entire document in memory in the form of a tree. The nature of this DOM tree will cause it to consume a lot more space in memory than the plain XML document itself. This makes DOM parsing extremely prone to denial-of-service abusage. A stream-based parser, however, uses an event-driven approach: the document is traversed in a linear fashion and triggers events when encountering certain characters. There is no need to keep the whole document in memory. The implementation of stream-based parsers has become more widespread as the weaknesses of DOM became more prominent. For example, when Apache moved from Axis to Axis2, stream processing was introduced, resulting in a web service implementation that is far more resilient to denial-of-service attacks. However, it is known that Rampart, the WS-Security module for Axis2, still constructs the whole document in memory [6]. This creates an important bottleneck for Axis2, since the initial DDoS vulnerabilities solved by opting for a stream-based parser are basically nullified when using WS-Security.

Jensen et al. [4] also introduce the concept of *schema hardening* as a defense strategy, which is based upon the research conducted by Gruschka et al. [7]. They state that proper schema validation could solve the threats to some extent: if only the necessary elements are allowed to be used in a SOAP request, abuse can be prevented or drastically limited. However, most schemas are not restrictive enough: often they allow an infinite amount of elements to be present, indicated by the "unbounded" keyword. Therefore, Gruschka et al. propose to *harden* the schema. This involves adding more restrictions, with respect to the specifics of the particular web service. Although their approach is effective at preventing and detecting abuse, the hardening is a manual and tedious task involving thorough analyses of the web service operations. This is not feasible in a cloud environment, where different web services from different developers can be deployed at any time.

Pinzón et al. propose S-MAS [8], a distributed multi-agent architecture for blocking malicious SOAP messages. They use CBR (Case Based Reasoning) as the main principle of their system. Features are extracted from the SOAP requests, which are used for anomaly detection. They use two stages of classification: first, a fast classification is conducted using a decision tree. The second stage implements an artificial neural network (ANN), which is trained with attack and normal requests. The system is adaptive in the sense that it requires manual classification and retraining if the system is undecided or erroneous decisions have been made.

Chonka et al. adapted their previous research into Cloud Traceback (CTB) [3]. This traceback system uses marking of SOAP messages in the SOAP header field. The marking enables them to identify the malicious source by tracing back after the web service has suffered from an attack. Additionally, they constructed Cloud Protector: a backwards propagation neural network to detect malicious SOAP requests.

Siddavatam et al. [9] propose testing for several features of XML requests in order to detect certain attacks, of which also DOS attacks. However, they use certain fixed thresholds, for example, max document size, that have to be configured by an administrator.

Menahem et al. [10] propose a probabilistic one-class classifier for detecting malicious patterns in XML documents. They approximate density functions of single attributes from their training set, which can produce a per-attribute normality score. Furthermore, they combine these scores via yet another univariate density-function model. This final density function produces the output for test data. A threshold determines the classification of the combined normality score derived from the test data.

As for HTTP DoS attacks, [11] proposes Threshold Based Kernel Level HTTP Filter (TBHF), a filtering system that counters malicious URL shortener services. After clicking a shortened URL, these services forward the user to the expected page. Additionally, they secretly open an iframe that cooperates in an attack by sending HTTP floods in the background. TBHF is a filter installed on the client machine that detects possible outgoing floods.

Unlike S-MAS, which intercepts requests on the network layer, the proposed system fully operates at the application layer by intercepting HTTP requests. Furthermore, outlier detection is used instead of classification with an ANN, which both Cloud Protector and S-MAS implemented. The defense system uses

Table 1
The vulnerabilities that were tested on Axis2.

Possible attacks	Confirmed
HTTP flooding	Yes
Coercive parsing	Yes
Oversized XML	Yes
Oversized encryption	Yes
WS-Address spoofing	Yes
Infinite key retrieval loop	No
Malicious transformations	No
Signature reference redirect	No
DTD entity expansion	No

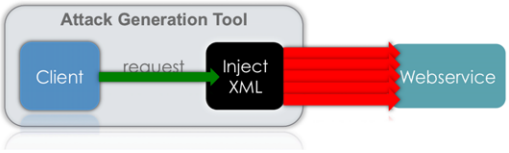

Fig. 1. The attack-generating tool.

this outlier detection to implicitly, dynamically and adaptively harden the schema, elaborating on the proposal of Gruschka and partly Siddavatam. Menahem et al. use Parzen Windows to detect anomalies; this paper finds this unfavorable regarding that Parzen Windows do not scale well with a large dataset. Also, this paper incorporates the concept of TBHF, but adjusted it to be placed at the server's end.

3. Attacks

First, an overview of all regarded attacks is given. Second, an attack-generating tool is presented. It is able to test the regarded attacks on a web service implementation, Axis2 in this case. Finally, the confirmed vulnerabilities are discussed.

3.1. Possible attacks

Several reports [12,13,4] have been made which address XML vulnerabilities facing web services. Recent HTTP attacks on government websites in Iran are discussed in [3].

The attacks noted in Table 1 were regarded and implemented in the attack-generating tool.

3.2. Attack-generating tool

In order to test these possible vulnerabilities, an attack-generating tool was developed. The tool was designed to take in a legitimate request to a web service and alter it into a message that exploits one of the listed vulnerabilities. This malicious message is then sent to the victim web service. The attack-generating tool also incorporates the possibility of flooding the web service with this message.

As seen in Fig. 1, the attack-generating tool consists of two parts: a legitimate client that produces a normal request and the application that injects that request with malicious XML.

All the stated vulnerabilities can be exploited using this tool, except for the attacks involving encryption or signatures. They were not included in this tool because the security procedures, such as signing, will prevent the injection tool from tampering with the message. It is necessary to generate legitimate WS-Security requests in order for the message to be fully processed. Therefore, it was opted to create the oversized encryption attack within a client itself. This malicious client created a request with an oversized encrypted body.

Unfortunately, a super encryption attack could not be tested. No information on how to compose super encryption with Rampart was found.

3.3. Vulnerability testing and confirmation

With the attack-generating tool, the possible vulnerabilities facing the availability of web services could be tested. The tests in this paper were focused on Apache Axis2, which is considered the defacto standard for open source web service deployment [14]. A WSO2 Application Server [15] was deployed. WSO2 5.1 comes with an installation of Axis2 1.6.2 [16] and Rampart 1.6.2 [17], the module that handles WS-Security.

Test attacks were launched on sample web services running on the Axis2 installation. Both single malicious requests, as well as flood attacks were conducted. If a significant amount of resource consumption (CPU and/or RAM memory) was recorded, the attack was considered successful. The confirmed vulnerabilities are marked accordingly in Table 1. The remaining reported vulnerabilities will still be regarded as potential threats. The detection of these will be incorporated in the defense tool, so if the vulnerabilities happen to surface in another situation, the defense tool will be capable of detecting these.

3.4. Discussion of confirmed vulnerabilities

3.4.1. HTTP flooding

In the case of web services or other resource intensive requests, HTTP flooding with legitimate requests may result in a denial-of-service because of the regular computations involved in completing a request. Additionally, the communication channel of the server might be exhausted.

3.4.2. Oversized XML

The attacker sends requests with a very large XML document (several megabytes in size), which can contain elements, attributes or namespaces with large names or content. DOM parsed documents increased with a factor 2–30 in memory, compared to the original XML document. We can distinguish various subtypes: XML Extra Long Names, XML Namespace Prefix Attack, XML Oversized Attribute Content, XML Oversized Attribute Count.

3.4.3. Coercive parsing

Similar to the oversize payload attack, a malformed XML will clog up resources. The difference is that coercive parsing is mostly targeted at CPU cycles, instead of memory consumption. This is achieved by incorporating many namespace declarations, large prefix or namespace URIs, a continuation of open tags, or simply very deeply nested XML structures.

3.4.4. Oversized encryption

When encryption and digital signatures are required in SOAP, Axis2 uses the Rampart module for WS-Security. As stated in Section 2, Rampart places the whole request in memory in order to decrypt or verify the signatures. This results in huge memory consumption compared to the initial message size.

3.4.5. WS-addressing spoofing

WS-Addressing allows you to specify a ReplyTo and a FaultTo address in the SOAP header. It tells the web service to send the SOAP reply or fault message to a different location than the original sender. As noted in [4], this feature can be abused in a reflective attack: the sender can specify the victim address in the request to the web service, forcing the service to send data to that victim. The concept is analogous to a DNS reflective attack [18]. Axis2 enables the use of WS-Addressing by default. Therefore, all Axis2 implementations are vulnerable to this abuse if they are not configured otherwise.

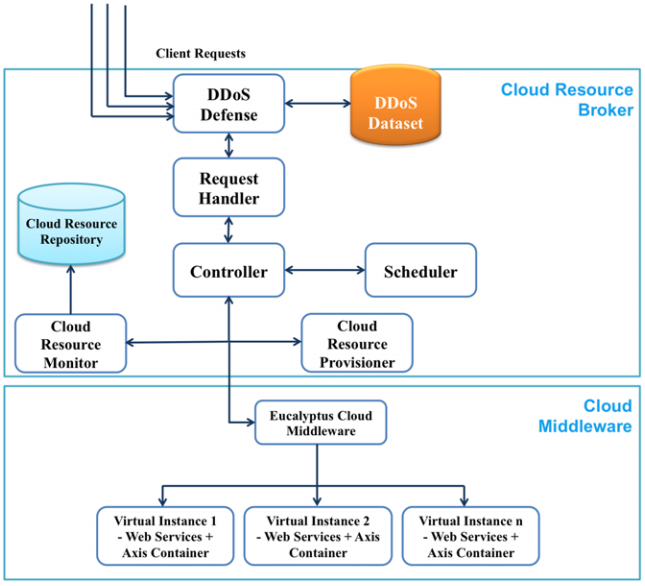

Fig. 2. A cloud resource broker architecture with the DDoS defense system in place.

4. Architecture

The following section discusses the architecture involved in implementing the proposed defense system in a cloud environment. First, the cloud architecture without any defense system is reviewed. Then, the secure architecture is introduced.

4.1. Cloud architecture

The common, unprotected cloud architecture contains the following entities:

- *Cloud providers:*
  The providers are the actual workhorses of the cloud environment. The cloud middleware dynamically deploys the virtual machines and web services on these entities.
- *Cloud broker:*
  The Cloud Broker listens to the initial request by the user. It keeps track of the available cloud providers and allocates the necessary resources for the user.
- *User:*
  A user is a client that connects to the Cloud Broker to request resources from the cloud environment. Eventually, it can interact with the web services deployed in the Cloud Providers. In this paper a user can be either of legitimate or malicious nature.

The main weak spot for the cloud architecture is the Cloud Broker [2,19]. The users have to consult this broker in order to receive the services they need. Rendering the broker unavailable has a fatal impact on the entire service of the cloud system. This single point-of-failure concept makes them the most crucial entities to defend. The effect of attacking a Cloud Provider or deployed virtual machine is less catastrophic. Although, protecting them could also be considered, depending on the overhead generated by the defense system.

4.2. Secured architecture

To secure the cloud architecture, a defense system is deployed as the first step in the cloud broker, as seen in Fig. 2. The DDoS Defense system analyzes the user application requests: if the application request is found to be malicious, it is simply rejected. Else, it passes the user application requirements to the Request Handler and the request is further processed routinely by the broker.

In the case of a distributed broker architecture, the defense system can be incorporated in all broker entities. For better results, datasets can be shared between the distributed instances of the defense system. This can be done by storing the dataset in a central location, which can be accessed by all defense system instances in the distributed architecture. However, a single point of failure should be avoided by keeping a local copy or, preferably, by using a distributed storage mechanism. The implementation of such an architecture is left as future work.

Because the proposed defense system focuses on HTTP and XML attacks, it operates on layer 7 of the OSI model. At this layer, IP-packets, TCP handshakes, etc. have already been processed and are not relevant anymore. Consequently, a simple packet sniffer cannot be used to detect and mitigate HTTP and XML DoS attacks. The proposed defense system incorporates a reverse HTTP proxy as the filter. This proxy will intercept all passing HTTP requests.

This filter should not interfere with the standard operation of the cloud architecture in any way. It is designed to be deployed requiring only minimal changes to an existing system. While operational, it functions entirely in the background, hidden from all the existing components. For example, the user is not aware of this extra filtering step; he simply connects to the web service/broker using the address of the defense system. The real destination address can simply remain hidden to the user. However, to be completely safe, the web service/broker should be configured to only listen to requests coming from the defense system. The filter can be deployed on the same operating system as the web service/broker or on a dedicated (virtual) machine. The system is completely flexible and transparent. The only practical requirement for implementing this system is that the two components have a unique IP address and port combination so that they can be distinguished from each other.

5. Filter design

The core concept of the filter is to determine a normal usage profile for each web service or web service operation. This profile is represented by Gaussian models, defined by means and standard deviations of several features, such as content-length, number of elements, nesting depth, longest element, attribute and namespace. These models are based on datasets constructed from the logged features of previous requests. Using these models, outliers can be detected to distinguish abnormal requests that are sent to a web service. In addition, the system also aims to cover HTTP request limiting, SOAP Action and WS-Addressing spoofing detection.

A main design aspect of the proposed system is to reduce computational complexity: the impact must be kept minimal in order to prevent affection of the overall experienced response time. Moreover, it enables the defense system to scale and withstand a denial-of-service attack itself.

For fast interception of request, LittleProxy [20] was used. It is based on Netty [21], an asynchronous event-driven network application framework, designed for high performance.

As the datasets grow, the generation of models can become a crucial factor for the computation time. Therefore, it was opted that the proposed system pre-generates all models when the system is started. Afterwards, during filtering, each feature model can be accessed in constant time, independent of how big the dataset has grown. However, the system also allows for on-the-fly recalculating of the models if an immediate update is considered necessary. For the live filtering of requests, performance has been considered as well. A request has to pass through two separate phases: the first phase only processes the HTTP header of the

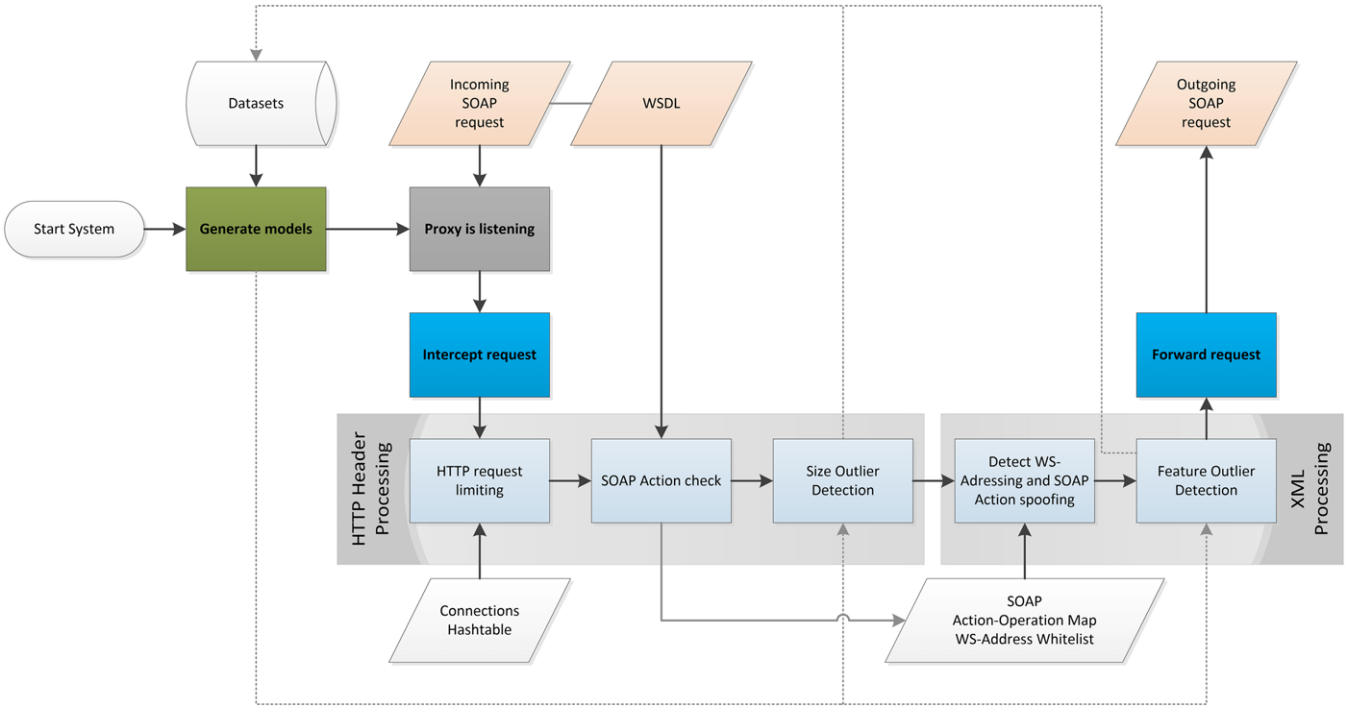

Fig. 3. Flowchart of the filter.

request. This allows for fast detection of HTTP floods, unexisting SOAP Action usage and content-length outliers. Only after passing these tests, the actual XML content will be parsed and inspected. This second phase incorporates the SOAP feature outlier detection and finds any SOAP Action or WS-Addressing spoofing. A flowchart of the design of the filter is depicted in Fig. 3.

In the following section, the separate elements of the filter will be discussed. Each process listed in the systems' flowchart will be handled chronologically.

5.1. Initialization

During initialization, the normal profile models are calculated from the datasets. Afterwards, the proxy is launched and starts listening for requests.

5.1.1. Model generation

In order to detect outliers, a representation of the normal profile is necessary. The system uses a normal distribution (Gaussian) to model the normal profile. These models are generated when the system is started. First, the file containing the complete dataset will be retrieved. Next, all entries belonging to a particular SOAP Action are grouped together in smaller datasets. Finally, for each feature in each of these datasets, a Gaussian Model (GM) is generated, by calculating the means and standard deviations. These are stored in an array that can be retrieved fast from a hashtable by specifying the particular SOAP Action as key.

5.2. Phase 1: HTTP header inspection

5.2.1. HTTP flood prevention

To prevent HTTP flooding, the proposed system will limit the amount of requests before doing any further processing. The system only allows a single request from a client within a specified timespan, Δt. The default is set to 1000 ms, but can be adjusted by an administrator. To achieve this limiting, a hashtable is used to keep track of the connected addresses and the last time they made a request to the web server.

5.2.2. SOAP action check

In the HTTP header of a SOAP request, a "SOAP Action" field might be present. This field specifies which SOAP operation is called in the body of the SOAP request:

```
POST http://example.com/Soap12/ HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "urn:logout"
```

The SOAP Action header can be used to determine the requested operation without having to inspect the actual XML body. The proposed system observes this field for two purposes: first of all, it is used to distinguish between different operation calls, so appropriate and accurate outlier detection can be done on the message size. Second, attackers might spoof the SOAP Action header in order to mislead the victim, as noted in [4]. Moreover, spoofing can mislead our system as well. Therefore, the system performs various checks to determine if a correct SOAP Action is used.

If a SOAP Action enters the filter that has never been observed before, the system will find and access the Web Services Definition Language (WSDL) belonging to that particular web service. It will traverse the document and map all specified SOAP Actions to their corresponding SOAP operation. An example of such a statement in a WSDL file is as follows:

```
<operation name="logoutUser">
<soap:operation soapAction="urn:logout"/>
```

Later, in the second phase, when the actual XML is being processed, a mismatch between the SOAP Action and operation can easily be detected by consulting this mapping. Now, during this phase, it can only be determined if the SOAP Action used actually existed within the WSDL.

5.2.3. Size outlier detection

The final step in the first phase is to detect any outlier regarding the size of the request. From the HTTP header, R_{size} is extracted from the content-length field. Next, the mean and standard deviation associated with $R_{SOAPAction}$ are retrieved from GM. Using these

values, the upper and lower bounds B are calculated. Here, the α parameter is introduced. This parameter specifies how many standard deviations the observed value is allowed to differ from the mean. Currently, the parameter is set in an ad-hoc manner to 3.5, which corresponds to the 99.95 percentile of the cases in the dataset, assuming that the dataset has a normal distribution. When the administrator has gathered a training set for his particular web services, a learning process can be applied to find the optimal value for the α parameter, effectively changing the strictness of the filter. In cloud and grid environments, where a trust value is given to users, the administrator may choose to incorporate this value to tune the α parameter.

Afterwards, the system checks if R_{size} exceeds the boundaries B; if so, the request R is dropped. Otherwise R is sent to the next phase and R_{size} can be appended to the dataset.

5.3. Phase 2: XML content inspection

In the second phase, the actual XML content of the request is processed. The XML is traversed as a stream using SAX [5]. The system extracts a collection of features from the content. These features are chosen in association with the discussed attacks in Section 3.1, effectively exposing every listed attack and constructing a meaningful representation of a normal request. Furthermore, the necessary information to detect spoofing attacks is extracted as well.

5.3.1. SOAP action and WS-addressing spoofing

After the XML content has been traversed and all the necessary information has been gathered, the system will first check if $R_{SOAPAction}$ specified in the HTTP header was spoofed. This is done by consulting the previously made mapping and comparing $R_{SOAPAction}$ to $R_{operation}$ called in the XML content. Next, the system will make sure that no illegal WS-Addressing requests are made. An external whitelist file is thereby consulted.

5.3.2. SOAP feature outlier detection

The final step in the filtering process is to evaluate each extracted feature to its corresponding GM. The calculations of the boundaries B for a single feature using the α parameter are completely analogous to those of the size outlier detection. If no feature is found to be an outlier, the features of R can be appended to the dataset. Finally, the proxy will forward the request to the destination web service for normal operation.

5.4. Sequence diagram

The full sequence diagram of the defense system can be seen in Fig. 4. The administrator starts the system and initiates training. At this stage, any request from the user will be accepted and appended to the dataset. Later, when training is turned off, the system is live and detects malicious messages with the models generated from the dataset.

6. Evaluation of the proposed system

6.1. Experimental setup

The experimental setup deployed in the Cloud Resource Broker (CRB) is shown in Fig. 5. The CRB is installed on a server with 4 CPUs; each CPU has a Quad Core Processor with 2 GHZ processor speed, 16 GB RAM memory and Cent OS 5.5 as the operating system. The hardware details of the Cloud resources configured in the Anna University Campus are shown in Table 2.

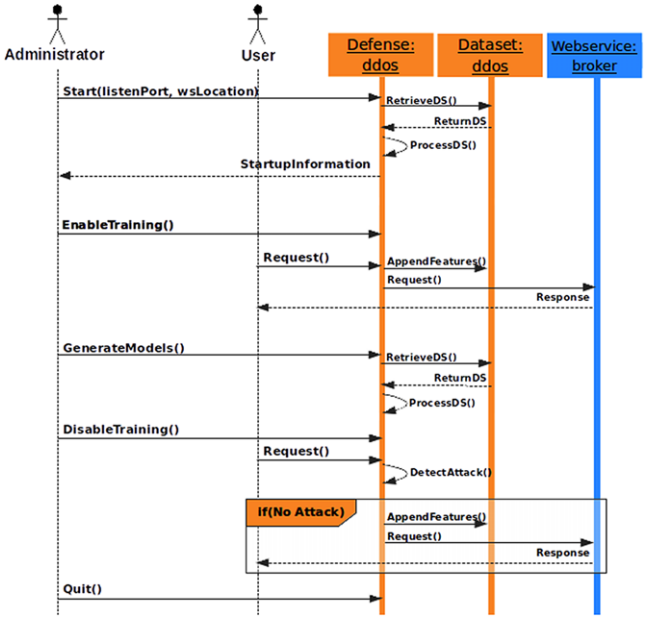

Fig. 4. The sequence diagram of the defense system.

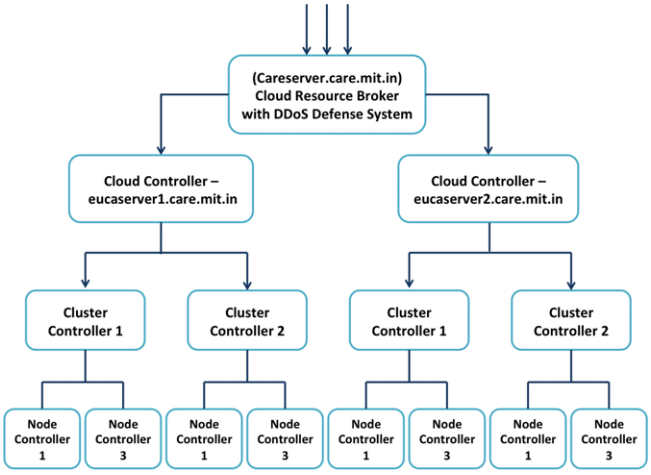

Fig. 5. The experimental setup.

Table 2
Hardware details.

Details	High-performance computing lab	CARE research lab
Processor	INTEL CORE DUO @ 3.0 GHZ	INTEL CORE DUO @ 3.0 GHZ
Architecture	i386	i386
Harddisk space	250 GB	250 GB
RAM	4 GB	3 GB
Kernel version	2.6.33.x (KVM)	2.6.18.x (Xen)

The eucalyptus 2.0.3 middleware is installed for managing the Cloud resources. The Eucalyptus based Cloud resources are configured with Cent OS 5.5 and Fedora 11 as the operating system in a mixed mode. The eucalyptus based Cloud resources have two cloud controllers. Each cloud controller is registered with two cluster controllers. The cluster controllers are attached to five to ten node controllers. The eucalyptus based Cloud resource is configured with Xen hypervisor, version 2.6.33.x and eucaserver2.care.mit.in is configured with Kernel Virtual Machine (KVM) hypervisor, version kvm-36. We have bundled the Cent OS 5.3 operating system image with the DDoS Defense System, WSO2

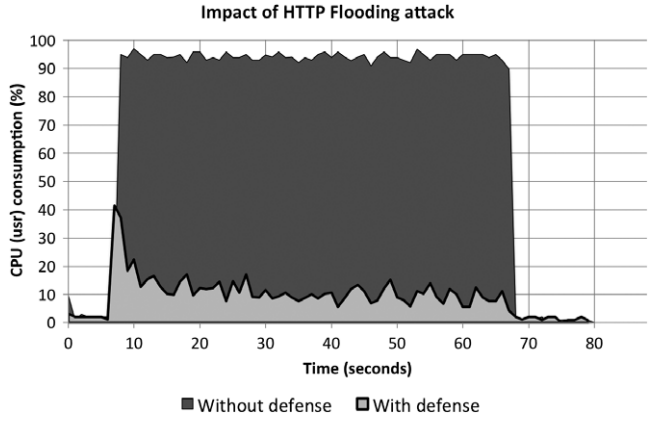

Fig. 6. Impact of a 60 s HTTP flooding attack.

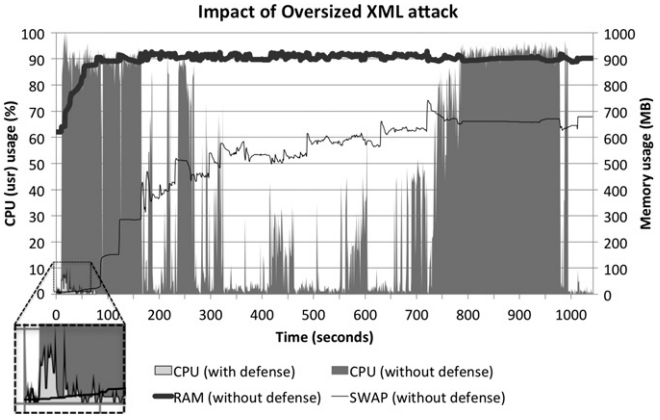

Fig. 7. Impact of an oversized XML attack: 10 messages of 12 MB.

5.1 application server and Axis2 1.6.2 container, deployed with sample web services.

Once this setup was operational, the proposed system was evaluated in two aspects: impact and performance. The former looks into the impact of the attacks and how well the system achieves mitigation. The latter looks at the extra response time that is introduced by using the system.

The Axis2 web service and the defense system were deployed together on a virtual machine running Ubuntu 12.10, with 1 GB of RAM and 1 CPU core of an Intel i7-3520M (2.90 GHz) to its disposal.

6.2. Impact and mitigation

These tests measure the impact of the different attacks stated in Section 3.4. Additionally, they show the effectiveness of the filter at mitigating them. Unless stated otherwise, a web service with minimal resource consumption is used during the attacks. This ensures that the measurements are hardly affected by the resource usage of the web service.

6.2.1. HTTP flooding attack

Throughout this attack, a web service with considerable computing time was flooded with legitimate requests. During 60 s, over 4900 requests were sent. As shown in Fig. 6, this clogged up all the CPU cycles (the remaining 5%–10% are used by the kernel). When the defense system is activated, only a small CPU spike can be noticed when the attack commences. The first request is accepted, but the following ones are rejected because they exceed the request per second limit. Moreover, after the system registers too many violations, it will blacklist the source address. Thereafter, only minimal computing of around 10% CPU power is necessary to drop the subsequent malicious flooding requests.

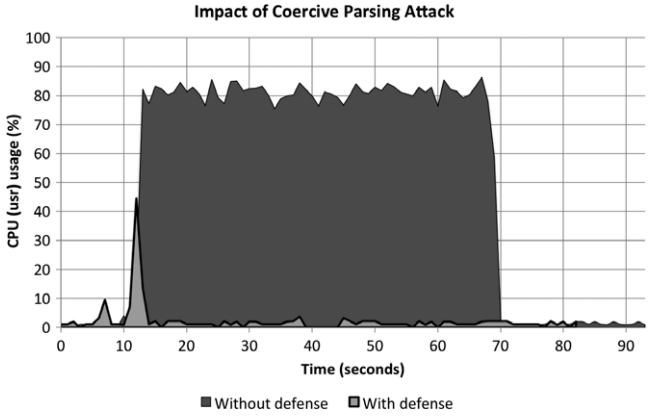

Fig. 8. Impact of a coercive parsing attack: 100 deeply nested messages.

6.2.2. Oversize XML attack

For the oversize XML attack, 10 messages of 12 MB with a long element name were sent to the web service. To prevent the HTTP flooding mechanism from detecting this attack, the attack-generating tool was set to wait 1000 ms between requests. The unprotected web server was severely suffering for over 15 min from these 10 requests. In Fig. 7, the CPU and memory usage is displayed.

The defense system handled the attack well; no extra memory consumption was noted. Only some small CPU spikes of around 10% were recorded for a period of 20 s, when it was blocking the attack.

6.2.3. Coercive parsing attack

For the coercive parsing attack, 100 messages of 210 kb each, containing a deeply nested XML structure, were sent to the web service. As explained in Section 3.4.3, this induces a lot of CPU consumption. To bypass the HTTP flooding detection of the defense system, a different source port was used every request.

In Fig. 8, it is shown that it took the unprotected web service about a minute of full CPU load to handle all these requests. The web service returned a SOAP fault message that reported a stack overflow error.

Although the requests are of a reasonable size, the feature outlier detection immediately finds the message suspicious because of the deep nesting structure. The defense system processes and drops them all in a matter of seconds, never saturating the CPU.

6.2.4. Oversize encryption attack

As stated in Section 3.4.4, encrypted messages consume a lot of additional memory while being processed by Rampart. In this attack, only a single encrypted SOAP request of 45 MB was sent to the web service. As seen in Fig. 9, the message is received at the 20 s mark. The RAM immediately shoots up and saturates. The rest of the memory has to be placed in swap space. This gradually increases to about 600 MB. During this period the machine freezes and hangs. All this time the CPU is full of wait cycles, waiting for the I/O. The system remains unstable indefinitely, as the memory consumption does not reduce over time.

The defense system, however, immediately discards the large encrypted message, as it detects the message size as an outlier. This process does not involve any noteworthy CPU or memory consumption.

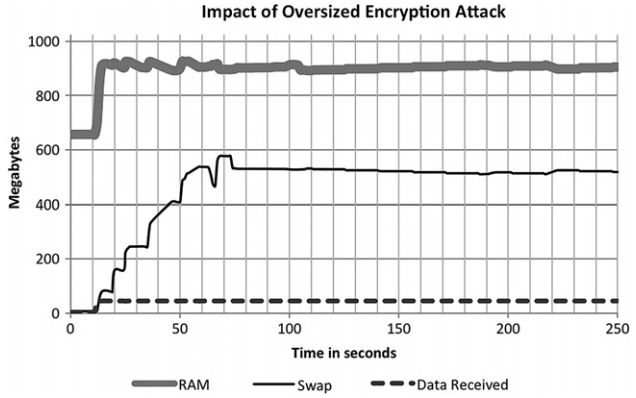

Fig. 9. Impact of an oversize encryption attack: a single 45 MB encrypted message.

Table 3
Load test: 2 requests per second. (Response time in ms).

	95% CI	99% CI	Average
Empty web service	2.93–3.07	2.91–3.09	3
Including defense system	4.12–5.88	3.84–6.16	5
Defense system impact	–	–	2

6.2.5. Spoofing attack

The defense system was able to detect spoofing of the SOAP Action in the HTTP header, or usage of *ReplyTo* addresses that were not on the whitelist. However, it should be noted that when using encrypted messages attack obfuscation can take place, as stated in [4].

6.3. Response time

Load tests were executed using SoapUI 4.5.1 [22] and LoadUI 2.6.6 [23] to measure the impact of the defense system on the total response of a legitimate request. Two different types of load tests were conducted: the first one sent 200 SOAP requests at a rate of 2 requests per second. The other one sent 10,000 SOAP requests, at 50 requests per second. These tests were executed both with and without the filter in place. This enables us to isolate the impact of the defense system. As before, a web service with minimal resource consumption is used. All requests were constructed so that they would pass through all stages of the defense system. This involved disabling the limit of requests per second, but whilst keeping all those computations in place. Consequently, worst case response times are recorded. Furthermore, it should be noted that the performance of the defense system is not correlated to the computational complexity of the actual web service: these tests try to isolate an absolute time delay induced by the computations of the defense system, by excluding the dummy web service response time and network delays. It is therefore meaningless to consider a relative overhead percentage.

6.3.1. Steady load

During 100 s, a SOAP request was sent out every 500 ms. As seen in Table 3, this resulted in an average response time of 3 ms for the unprotected web service. When the defense system was included in the chain, the total average increased to 5 ms, resulting in an average processing time for the defense system of 2 ms.

6.3.2. Higher load

This test involved sending 10,000 requests over a period of 200 s, effectively letting the web service process 50 requests per second. As listed in Table 4, the unprotected web service averaged again at a response time of 3 ms. After including the defense system, this increased to 9 ms, resulting in an average overhead of 6 ms.

Table 4
Load test: 50 requests per second. (Response time in ms).

	95% CI	99% CI	Average
Empty web service	2.99–3.01	2.99–3.01	3
Including defense system	8.40–9.60	8.21–9.80	9
Defense system impact	–	–	6

6.3.3. Comparison with alternative systems

For the S-MAS system [8], response time was measured in correlation to message size. Furthermore, by training the system, the response time was reduced gradually day by day. This resulted in averages that varied between 6 and 11 ms for a 100 kb message and between 25 and 45 ms for a 1075 kb message. For comparison, the same sizes of messages were used on the proposed defense system. The measurements resulted in an average of 3.6 ms for 100 kb messages and 20.6 ms for 1065 kb messages, effectively performing faster than S-MAS.

For Cloud Protector [3], specifics about their test messages are not given. However, the best noted result was 10 ms, and the worst 140 ms. Given that the proposed system achieved a response time of around 2 ms and up to 20.6 ms for messages of approximately 1 megabyte, it can be concluded that it most likely outperforms Cloud Protector in terms of speed.

7. Conclusion

It is clear that application-layer vulnerabilities impose an enormous risk for the availability of web services. The tested attacks have proven to clog up a web server with a minimum amount of attack resources. A distributed attack is not even necessary, as a single machine and even a single request have proven to successfully execute a denial-of-service attack.

To mitigate this risk, a defense system is proposed and implemented. The system tests concluded that it is effective at detecting and mitigating these attacks. Besides, the defense system incorporates detection of all reported vulnerabilities and might be able to detect still unknown attacks because of its normal request model. However, if new attack techniques using other request semantics should arise, additional features have to be implemented. Finally, the response time overhead that is introduced is minimal. The system effectively is faster than the considered alternatives.

After training the system, there were no wrongly detected messages during testing. However, this paper consciously does not include any false positive or false negative data, as a proper training and testing dataset is not available yet. This dataset should contain real-world legitimate requests to multiple diverse web services. Additionally, attack requests should be present. Only with a representative dataset, meaningful false positive and false negative statistics can be stated and used for comparison between alternatives.

8. Future work

Further integration into the cloud environment must be investigated. Now, the system has only been implemented to protect the cloud broker. The low overhead might enable it to be used to protect all web servers. Additionally, the use of a shared, distributed dataset, or even common models should be considered to improve the scalability and the effectiveness of the system deployed in a distributed environment. Furthermore, adjustment of the α parameter, according to the gathered training data and assigned trust value of the user, as stated in Section 5.2.3, can enhance the system.

Testing on other web service implementations, such as CXF and .NET, must be done to determine what (different) vulnerabilities might be present there.

As stated in the conclusion, a representative dataset needs to be made or found to determine the accuracy of the defense systems. This dataset could be similar in concept to the KDD CUP 1999 dataset for network intrusion detection [24].

Acknowledgments

I, Thomas Vissers, would like to thank the people from The Centre for Advanced Computing Research and Education (CARE) at Anna University for making this research possible. Furthermore, I would like to express my gratitude towards Karthikeyan Panneerselvam, Avinash Nidhi and Shyam Narayan for their invaluable hospitality, and towards family and friends for their support.

We thank the reviewers for their valuable comments.

References

[1] A.S. Nitin Singh Chauhuan, Crypthography and cloud security challenges, CSI Commun. (2013).
[2] B. Cha, J. Kim, Study of multistage anomaly detection for secured cloud computing resources in future internet, in: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), IEEE, 2011, pp. 1046–1050.
[3] A. Chonka, Y. Xiang, W. Zhou, A. Bonti, Cloud security defence to protect cloud computing against HTTP-DoS and XML-DoS attacks, J. Netw. Comput. Appl. 34 (4) (2011) 1097–1107.
[4] M. Jensen, N. Gruschka, R. Herkenhöner, A survey of attacks on web services, Comput. Sci.-Res. Dev. 24 (4) (2009) 185–197.
[5] About SAX, Apr. 2004. [Online]. Available: http://sax.sourceforge.net/.
[6] D. Sosnoski, Java web services: Metro vs. Axis2 performance, Jan. 2010. [Online]. Available: https://www.ibm.com/developerworks/java/library/j-jws11/.
[7] N. Gruschka, N. Luttenberger, Protecting web services from DoS attacks by SOAP message validation, Secur. Priv. Dyn. Environ. (2006) 171–182.
[8] C.I. Pinzón, J. Bajo, J.F. De Paz, J.M. Corchado, S-MAS: an adaptive hierarchical distributed multiagent architecture for blocking malicious SOAP messages within web services environments, Expert Syst. Appl. (2010).
[9] I. Siddavatam, J. Gadge, Comprehensive test mechanism to detect attack on web services, in: 16th IEEE International Conference on Networks, 2008, ICON 2008, IEEE, 2008, pp. 1–6.
[10] E. Menahem, A. Schclar, L. Rokach, Y. Elovici, Securing your transactions: detecting anomalous patterns in XML documents, 2012. arXiv preprint arXiv:1209.1797.
[11] A. Mohamed Ibrahim, L. George, K. Govind, S. Selvakumar, Threshold based kernel level HTTP filter (TBHF) for DDoS mitigation, 2012.
[12] A. Falkenberg, Attack categorisation by violated security objective availability, Aug. 2010. [Online]. Available: http://clawslab.nds.rub.de/wiki/index.php/Category:Attack_Categorisation_By_Violated_Security_Objective_Availability.
[13] F. Hirsch, P. Datta, XML signature best practices, Jan. 2010. [Online]. Available: http://www.w3.org/TR/xmldsig-bestpractices/.
[14] Faq on web services and Apache Axis2, Feb. 2011. [Online]. Available: http://www.packtpub.com/article/faq-on-web-services-and-apache-axis2.
[15] WSO2 application server, Mar. 2013. [Online]. Available: http://wso2.com/products/application-server/.
[16] Apache Axis2, Apr. 2012. [Online]. Available: http://axis.apache.org/axis2/java/core/.
[17] Apache Rampart—Axis2 security module, Apr. 2012. [Online]. Available: http://axis.apache.org/axis2/java/rampart/.
[18] CERT, DNS amplification attacks and open DNS resolvers, Apr. 2013. [Online]. Available: https://www.cert.be/pro/node/16010.
[19] P. Varalakshmi, S.T. Selvi, Thwarting DDoS attacks in grid using information divergence, Future Gener. Comput. Syst. 29 (1) (2013) 429–441.
[20] A. Fisk, LittleProxy, Mar. 2013. [Online]. Available: http://www.littleshoot.org/littleproxy/.
[21] The Netty Project, Netty, May 2013. [Online]. Available: http://netty.io/.
[22] Smartbear Software, Soapui, Mar. 2013. [Online]. Available: http://www.soapui.org/.
[23] Smartbear Software, Loadui, Jan. 2014. [Online]. Available: http://www.loadui.org/.
[24] KDD, KDD CUP 1999 data, Oct. 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

Thomas Vissers is a Master of Applied Engineering: Electronics—ICT at the Artesis University College Antwerp, Belgium. For his Masters thesis, he conducted research at the Centre for Advanced Computing Research and Education (CARE) at Anna University, India. Recently, he started working as a security researcher at Distrinet, KU Leuven.

Thamarai Selvi Somasundaram is working as Professor and Dean in MIT Campus, Anna University, Chennai, India. She received her Ph.D. in Computer Science and Engineering in Manonmaniam Sundaranar University, Tirunelveli. Her area of interest includes Artificial Intelligence, Neural Networks, Grid Computing, Grid Scheduling, Semantic Technology, Virtualization and Cloud Computing. She is the Principal Investigator for the CARE project funded by Department of Information Technology, Ministry of Communication and Information Technology, New Delhi. She is the reviewer for Future Generation Computing Systems (FGCS), Journal of Grid Computing, Journal of Parallel and Distributed Computing, Transactions on Mobile Computing and IEEE Transactions on Data and Knowledge Engineering.

Luc Pieters is a professor of communication networks and head of the department of Applied Engineering: Electronics—ICT at the Artesis University College Antwerp, Belgium. He completed his civil engineer studies in electronics at the VU Brussel in 1975. In the initial phase of the IMEC he took part in the training of chip designer. He obtained a special degree in informatics at the VU Brussel in 1984. He was also a member of the Academic Staff of the Postacademic Programme Telecommunications and Telematics at the University of Antwerp. He organized a 12 weeks TACIS training session for staff members of MMT in Moscow. He was coordinator–contractor of a Tempus project that enhanced the post-academic degree for telecom engineers at the Tashkent Electrotechnical University of Telecommunications.

Kannan Govindarajan has received the B.E. degree in Information Technology from Bharathidasan University. He has completed his M.S. (By Research) degree in computer science and engineering from Anna University, Chennai in the area of Grid Scheduling in Virtualized Grid Environment. He is currently doing his Ph.D. at Anna University. He has seven years of research and development experience. His area of interest includes Grid Computing, Grid Scheduling, Semantic Technology, Virtualization and Cloud Computing. He is currently working as a Senior Research Associate in CARE, MIT Campus, Anna University. In 2012, he had been invited to visit Athabasca University, Canada, for research interaction and development.

Peter Hellinckx obtained his Master in Computer Science and his Ph.D. in Science both from the University of Antwerp in 2002 and 2008, respectively. His master thesis acted on the interpretation of 2D sketches. His Ph.D. was titled "The evolution towards Desktop Grid systems, problems and solutions". In 2009 he joined TERA-Labs at the Karel de Grote University College where he became a senior researcher and responsible for the distributed computing research group. In 2013 he became a professor with the faculty of applied engineering (Electronics-ICT) at the University of Antwerp and joined CoSys-Lab. Here he is responsible for the distributed computing research team.