Mitigating DoS to Authenticated Cloud REST APIs

Régio A. Michelin, Avelino F. Zorzo, Cesar A. De Rose
Computer Science School
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre - Brazil
regio.michelin@acad.pucrs.br, avelino.zorzo@pucrs.br, cesar.derose@pucrs.br

Abstract—**Systems available on the Internet are day-by-day targets of Denial of Service (DoS) attacks. These attacks can leave a system with high response time or even make it unresponsive. A DoS attack can be executed at the network level, just by exploiting communication protocols weakness, or at application level, by exploiting implementation issues. Based on this scenario, this article presents a mechanism for mitigating DoS attacks aimed at exploiting REST applications using authentication tokens. This mitigation is based on the client behaviour, where it can be classified as possible malicious client. Our results show a response time decrease of 36% during an attack scenario applied to a cloud management system.**

Keywords-component; DoS; security; REST; cloud

I. INTRODUCTION

Nowadays, more and more systems become available on the Internet facilitating their exposure to several different types of attack. These attacks are intended to steal some information, deploy malicious code and even to make a system slow to respond, or worst, to become completely offline. This last kind of attack is called Denial of Service (DoS) attack, and its main goal is to bring a whole system offline, or at least make it very slow [1]. In order for the attack to achieve its goal, it consumes all computer resources like network bandwidth, CPU cycles or storage space. Once the system is compromised, legitimate clients are not able to have theirs requests responded.

When a malicious user is able to consume all computer resources from its target, and make the computer system unavailable for legitimate users, the attacker, in some cases, uses this DoS attack to perform extortion on their target. Recently the Feedly web site [2], which is a RSS feed aggregator, was victim of a powerful DoS attack that consumed its server's bandwidth and legitimate users were not able to access it. During the DoS, attackers contacted the web site owners asking for ransom to stop the attack.

The DoS usage increased in last years due to the increase of availability of services on the Internet, which was facilitated by the grow of cloud computing. A system on the cloud consists of physical or virtual machines that can be rented by developers that pay only for what they use. So, from the developers perspective, they do not need to worry with cloud infrastructure because its management is delegated to the cloud company (this business model is called Infrastructure as a Service - IaaS) [3]. Developers hove to be only concerned on writing the application service.

Many cloud companies grow using this cloud model, which became popular due to the advances of virtualization technology. Virtualization allows several virtual machines to share the same hardware. These virtual machines are, usually, managed by a software called hypervisor [4], *e.g.* VMWare [5], Hyper-V [6], XEN [7] and KVM [8].

However, a hypervisor is not enough to create a cloud, it is also necessary to include a system responsible to orchestrate the usage of several different hypervisors, physical computer allocation, storages, etc. Also, this system must provide a self-service interface to allow cloud customers to manage their own virtual machines [9]. Examples of Cloud Management Systems (CMS) responsible to provide this kind of feature are, for example, OpenStack [10], CloudStack [11] or Eucalyptus [12].

On the application level, a CMS provides several ways to allow cloud customers to interconnect their own systems and to manage virtual machines and services that will run on the cloud [13]. One way for doing that is through REST (Representational State Transfer) calls [14]. REST is an abstraction architecture that allows distributed systems to communicate over networks. However, once REST is available to customers, the cloud company must consider how it will be used to avoid that an attacker compromises the whole cloud system, which will impact several different cloud clients.

One strategy to avoid damage to a CMS is to use an authentication mechanism; hence only authenticated users will be allowed to perform operations using REST [15]. This authentication can be performed through user name and password, and upon a correct pair of user name and password, the CMS generates a token that will be used to allow the user to access REST operations. This kind of authentication is provided by different CMSs, but due to the way authentication is provided by the CMS, it is possible to explore that for DoS attacks.

Therefore, this work proposes a mechanism to mitigate DoS that attack REST calls in CMSs. This will be achieved by analysing client requests performed through REST calls, and based on the client information and behaviour, *i.e.* whether it is a legitimate client or not, to block REST calls. This mechanism is based on the client IP and a timed control queue. As a case study, we analysed the Keystone [16] component of OpenStack, which is the module responsible for identity management. Although we have applied our solution to a CMS, we believe that our solution can be applied to any application on the Internet that uses REST APIs.

This paper is organized as follows. Section II presents related work. Section III describes how the problem was observed in REST that relies on authentication tokens. Section IV proposes a solution to mitigate the DoS attacks. Section V shows our solution applied to OpenStack Keystone module. Section VI presents our conclusions and future work.

II. Related Work

Lu [13] work presents an empirical study related to cloud APIs. In his work, Lu analyses the Amazon Elastic Compute Cloud (EC2) APIs, and shows a quantitative classification related to its API. The majority of cases that cause the API failure are related to the call being unresponsive. There are also a significant portion of cases in which they brought the system to provide slow response time.

Kargl [17] studied the first DoS as well as its change to the DDoS (Distributed DoS), performed through several machines infected by daemons that allow an attacker to remotely control the machine. To defend a system against these attacks, he proposes a Linux kernel change that includes a mechanism to route different packages through round robin and last connection. This change basically creates a load balancing in order to properly distribute client requests among servers. Kargl solution works in a network level, and once the package is received, it is validated and if the client is in a suspect queue, the package is dropped.

The research performed by Beitollahi [18] presents a DoS defense mechanism operating at application level. In his work, Beitollahi creates a mechanism that attributes different points to each received connection, based on connection history and statistics. However, this solution is applied only if the client is a human, because when it detects a suspect behaviour it sends a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) to the client. However, since in our scenario the client will be other system interacting through REST, Beitollahi's solution cannot be applied.

A collaborative defense system against DoS is proposed by Tariq [19]. This defense system is deployed in several nodes over a network, so when a node detects any malicious traffic, it sends a sign to other nodes and then the malicious traffic is filtered, avoiding it to reach the target system. This control and analysis is performed by collecting packages in a time frame window, and the collected data is compared with known DoS behaviours. This research works at network level, so for application level attacks, which create a valid connection, it will not work properly allowing the malicious traffic to hit its target.

III. DoS in REST API

A. *DoS Taxonomy*

DoS attacks consist basically in consuming all system available resources. The attack target goals are bandwidth, when this attack is performed at network level, and CPU and storage, when the attack is performed at application level. In the latter case, the attack usually is more complex due to the need to create many valid requests to its target, making this attack much more expensive than a regular network level DoS attack. Thus the attack can vary based on its characteristics.

The different characteristics allow the DoS attack to be classified in different ways, *i.e.* it can be executed in different levels from network to application level. Mirkovic [20] presents a taxonomy to classify several different types of DoS attacks, as well as defense mechanisms. Actually, Mirkovic classifies Distributed DoS (DDoS), *i.e.* attacks that are performed by several different clients aiming a single target system. Fig. 1 presents a simplified version of Mirkovic DDoS attack classification.

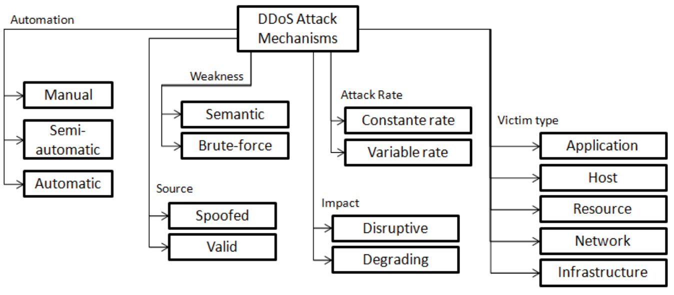

Figure 1. Taxonomy of DDoS attacks adapted from Mirkovic [20] work

As shown in Fig. 1, the source is used to evaluate if the attack is performed from a valid IP address, normally when the attack targets a specific application, or spoofed IP address, applied to create more noise during attack to avoid its detection. The exploited weakness can be semantic when the attack targets a specific protocol implementation bug or even a feature. During the attack, the number of packages sent can be used to classify it to a constant or random rate. Depending on the attack characteristics, it will let the target system slow or, in the worst case, completely offline, not allowing legitimate clients to have their valid requests answered.

B. *Authentication on REST*

Nowadays, as mentioned before, many requests performed over Internet are based on the architectural style known as REST [14]. REST calls are very important part to consider when designing a system over the Internet or cloud, because through this type of calls, different operations are exposed and many different applications can be integrated through them. Once REST calls are used, services will be available through the network to be accessed by users or systems. At this point, anyone with network access is able to perform a call to a REST operation, *i.e.* any malicious user (attacker) can use this exposure to try to compromise the system, for example, to perform a DoS.

The exposed REST services can be divided into services that do not require any authentication, *i.e.* the user does not need to provide a user name and password or token to execute an operation; and services that require an authentication mechanism, which usually is performed by using user name and password.

Fig. 2 shows the steps performed during a typical REST operation that uses an authentication mechanism. First, a client accesses the system address sending the user name and password (1). This is performed, usually, through a secure connection. The system validates this information and, if the user name and password are correct, a token is generated and stored in a database (2). After that, that token is returned to the user (3). Therefore, the user does not need to send the user name and password again every time a REST call is performed. So every time the user wants to perform an operation, the desired operation with the generated token is sent to the service (4). Once the token is received by the REST service, it must verify whether the user is allowed to execute the operation or not. This validation is performed consulting a database (5) in order to identify the user's permission level. If the token exists and the user has enough privileges, the operation is executed.

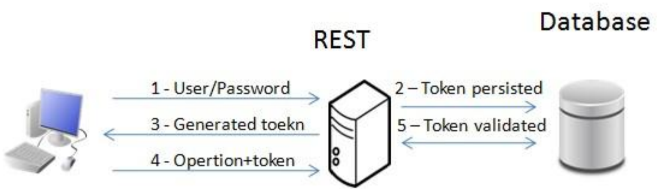

Figure 2. Regular REST flow considering authentication

C. Performing the DoS attack

REST services allow a client to execute several different operations that provide information about the system, for example, list of virtual machines, list of system tenant users, etc. If the client is not authenticated before executing these operations, the system can be compromised. Even though the authentication prevents a malicious user to access vital information about the system, token validation action can be a very attractive target for malicious users, because they can, continuously, send invalid tokens to try to overload that system. This might happen since the system will have to validate each invalid token that is submitted.

Basically, the service overload problem happens because upon receiving each request, the application has to check the database (or something similar to a database, *i.e.* a storage that contains valid tokens) in order to identify if the received token is valid and what are the operations the owner of that token can execute. This query may be time consuming depending on the type of access to the database. Traditional detection and traffic block mechanisms for DoS are not applied to this scenario, because all the incoming traffic (on the network level) is valid, *e.g.* the XML (eXtensible Markup Language) or JSON (JavaScript Object Notation) contents sent through REST calls are valid, only the token information is invalid. Usually, most DoS defense mechanisms work at network level, which consider only the information inside each network packet. Even some works that detect the malicious behaviour at application level, normally consider that on the other side of the connection there is a human user, not a system [18] (see Section II).

Fig. 3 presents the scenario where several malicious users are sending malicious traffic in order to consume the CPU target system, and then leading the legitimate client responses being delayed or even denied.

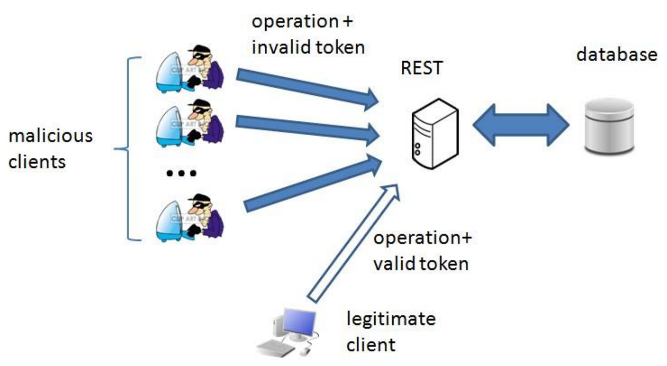

Figure 3. Attack scenario to a sytem with REST calls

IV. Proposed Solution

As mentioned in the problem described in Section III, the service overload leads to resource depletion when the DoS attack is executed. As mentioned in Section II, there are several different DoS attacks based on different attack aspects (see Fig. 1). Based on that taxonomy, this work focus on attacks that are automatic (automation), based on brute force (weakness), constant (attack rate), where the attackers have valid IPs (source), and their target is an application (victim type). The impact of the attacks can start as degrading, but can be disruptive. The proposed defense mechanism is reactive based on anomaly detection, running on the target computer, which takes an action based on the attacker agent identification.

Based on the presented problem (see Section III) and the above classification, we propose a defense mechanism at application level in order to minimize the unnecessary validation of tokens, avoiding the system to query the database. This implementation assumes that a legitimate client will not flood the system with invalid tokens.

Our mechanism works in two modes: monitoring and filtering. During monitoring, our mechanism verifies whether the system is being stressed or not, for example, when the CPU is overloaded, *i.e.* more than 70% of usage (this can be set with a different percentage depending on the type of system in which our mechanism is applied to). If the CPU is not overloaded, it takes the requests from each client, and if it detects that the requests contain invalid tokens, it marks this client as a probable attacker and includes this client in a gray list. When the system is overloaded, then our mechanism moves from monitoring mode to filtering mode, in which the gray list becomes a black list and clients that are in this list have their REST calls dropped. Therefore, any request performed by a client that is in the black list will be discarded as soon as it is received. Our mechanism gets back to monitoring mode again when the system is not overloaded, for example, CPU usage is 30%. Notice that there is a difference between how we consider whether the system is overloaded or not. This is performed to avoid our mechanism to change modes too frequently. Imagine the situation in which our mechanism changes from monitoring mode to filtering mode and right after it starts to drop REST packets, the CPU usage of the system, for example, becomes 69%. If we considered that the system is not overloaded anymore and we move our mechanism to monitoring mode again, we stop dropping REST packets, but in the next instant of time, the CPU usage could become 70% again, so our system would move again to filtering mode. This process could keep changing while the attack takes place, basically keeping the CPU usage at a peak that was not necessary if we dropped packets from the attackers. This would not make the system disruptive, but it would keep the system working in a degraded mode.

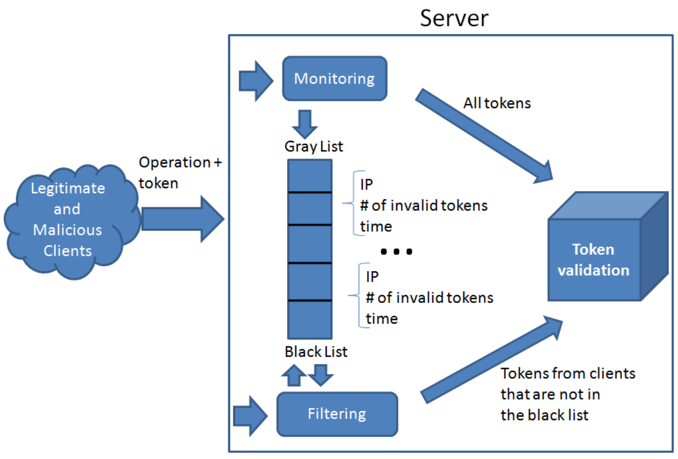

Figure 4. Client control solution achicteture

Fig. 4 shows our solution architecture, in which legitimate and malicious clients send REST operations and tokens. Notice that when our mechanism is in filtering mode, some invalid tokens will be verified if the clients are not in the black list. When this happens, during filtering mode, our mechanism includes those clients in the black list and their REST calls are also dropped. Requests from clients that are in the black list will be dropped for a period of time (window). After that window, our mechanism allows one request to be verified to check if the client now has a valid token. This may happen when a legitimate client sent invalid tokens (expired for example) and was included in the gray/black list, but after some time got a valid token. If we did not do that, legitimate clients could be blocked forever. If the client is malicious and keeps sending invalid tokens, then the time they are blocked (window) is increased.

V. Case Study: Keystone Module of OpenStack

In order to evaluate our proposed solution, we applied our mechanism in an open source CMS, *i.e.* OpenStack, which is composed by the following components: Cinder (responsible for controlling block storages for the virtual machines), Glance (responsible for managing operating system images), Swift (controls object storages), Neutron (responsible for network management), Horizon (provides the graphical user interface), Nova (orchestrate all OpenStack components), and Keystone (responsible for identity management). These different modules communicate through REST calls. For example, when a user accesses the cloud via the Horizon module, a token is generated by the Keystone module. Every time a client wants to execute a new operation, for example, to create a new virtual machine or to list the available disks, a REST call is made to the Horizon module that sends this call to the corresponding module, *i.e.* Nova or Cinder. These modules will send a REST call to Keystone to verify whether the token is valid or not.

As mentioned above, OpenStack modules communicate among them through REST calls (In the Fig. 5, arrows represent these calls).

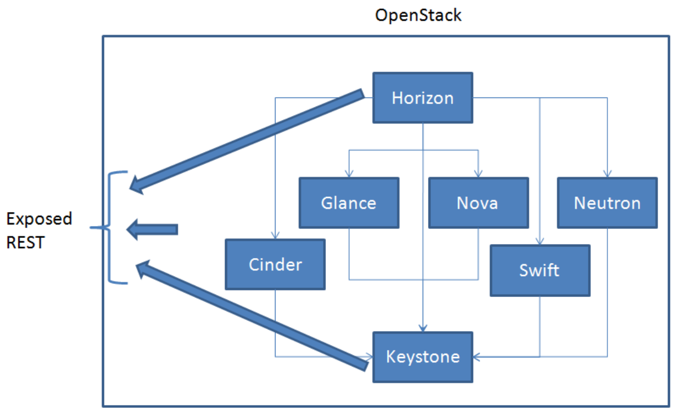

Figure 5. Simplified OpenStack archicteture

Fig. 5 shows a simplified OpenStack version. As can be seen in the figure, the Keystone module is central to the whole functioning of OpenStack, since it is responsible for the identity management. Therefore, Keystone is contacted by all modules to verify whether an operation can be executed or not. This is done because Keystone is responsible for storing all valid tokens and for checking if an arriving token is valid or not. Hence, Keystone is a good target for malicious users that want to overload the whole infrastructure provided by OpenStack. Once this user can get access to Keystone REST, he can send multiple requests containing invalid tokens, and this will force the component to consult the database, overloading the system.

In order to avoid Keystone REST exposure, during OpenStack installation, it is possible to configure the Linux firewall to block any call to Keystone REST. Hence, only users with access to the OpenStack management network are allowed to access the REST API. Despite the firewall protection, in some situations, like when a module has to be integrated with third party software, this REST must be exposed. Therefore, this default installation must be changed in order to allow access to the Keystone REST from different networks. In this work we assume the situation in which REST has to be exposed.

Our experiment focuses on two Keystone REST operations to be stressed, *i.e.* simulating an attack. After the attack is performed, we retrieve the requests response time to verify what the impact on the Keystone module is. The operations that are called are: token generation, where we assume that a real system user is calling the REST sending a valid user and password; and token validation through tenant list operation, so this operation requires a token to list all system tenants. Fig. 6 shows the time difference when both RESTs where overloaded. The experiment executes three different scenarios: i) clients start calling tenant list REST operation sending invalid tokens; ii) the same operation but now in a stress scenario where all clients are sending valid tokens; and, iii) stress situation with clients sending valid users and passwords but calling the token generation operation. In the three execution scenarios, our system started in an idle state, and the number of clients sending requests increased until 180000 requests. As can be seen in Fig. 6, response time degrades significantly faster for invalid token validation than for token generation or even for valid token validation.

A. Results Analysis

Since all cloud system were built on top of virtualization concept (and using different hypervisors), we chose to run our experiment on a virtualized infrastructure. The virtualization system used was VMware workstation 10.0.1 running on an Intel Core i5-4570@3.20GHz platform, with 16GB of RAM bus DDR3 1600 and Microsoft Windows 7 Professional 64 bits Operating System. The virtual machine where OpenStack Grizzly was running uses 4GB of RAM and 2 processors using Linux Ubuntu Server 14.04. The virtual machines for the client have 2GB of RAM and 1 processor running on Linux Ubuntu Server 14.04. The client virtual machines are used for starting the attack scripts.

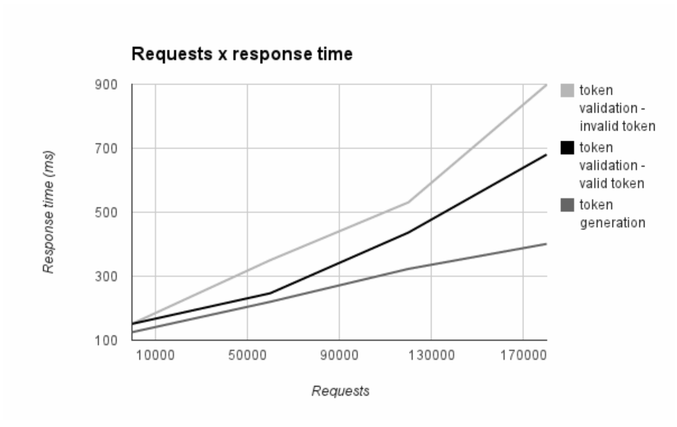

Figure 6. Response time increases when component is stressed

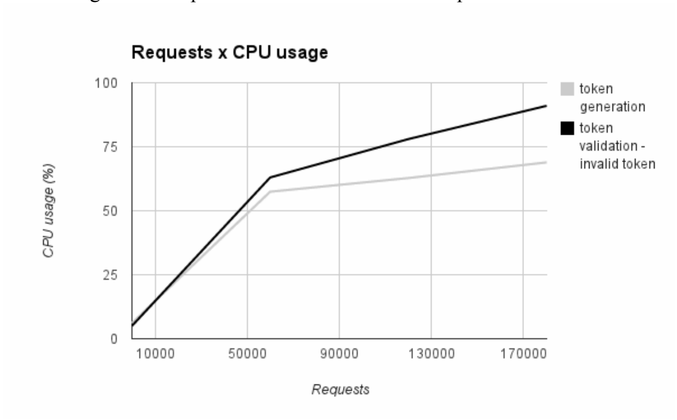

Figure 7. CPU usage during attack

Fig. 6 shows the response time, in milliseconds, for both REST operations (token validation and generation). The system handling 180000 invalid tokens requests for the Keystone REST tenant list operation, would take 898 milliseconds (error 10%) to respond to a valid client. The same system validating valid tokens for 180000 valid requests, would take 425 milliseconds (error 16%). On the other hand, while the system was idle, this token generation time was 151 milliseconds (error 11%). Besides this response time growth, the processor usage also increased, reaching around 90%. Fig. 7 shows this use and also that token validation is the operation that demands more processor usage.

Once the problem was identified in the Keystone REST operation, our solution was deployed in order to analyse the received tokens. Running the system again with the same load of 180000 and performing the tenant list operation with an invalid token, the response time drops to 328 milliseconds (error 10%). This time is approximately 36% faster than the same scenario without our solution. Fig. 9 shows this time difference.

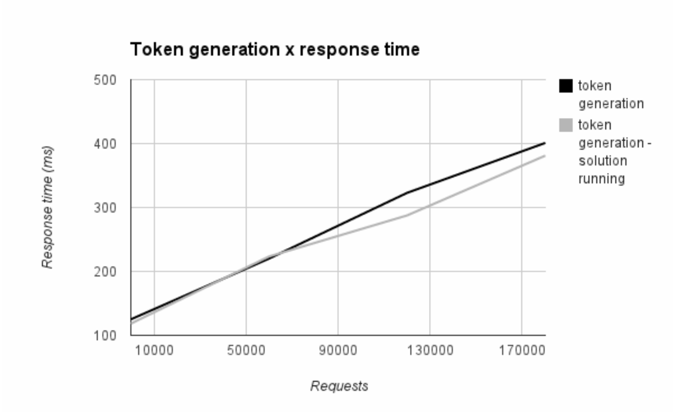

Figure 8. Response time during token generation

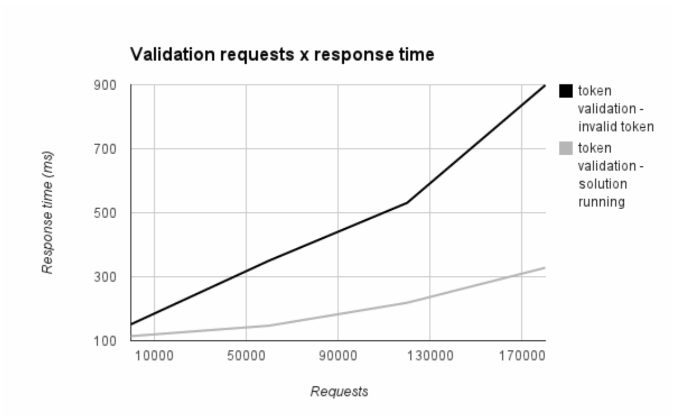

Figure 9. Response time with our solution running

Fig. 8 shows the token generation during a system stress scenario. We noticed that this response time varied from 401 milliseconds to 381 milliseconds considering an error of 15% to our samples. This behaviour happens since the proposed solution works only for handling the token validation operation. We also noticed the processor usage decreased when our solution is running. Fig. 10 shows the processor usage on the system when our solution was applied. Our solution was responsible for keeping its usage around 40%, against 90% usage when no solution was applied.

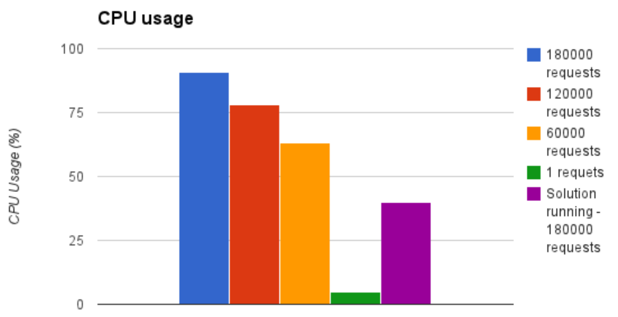

Figure 10. Processor usage with our solution running

VI. Conclusion

In this paper we presented a solution to avoid DoS attacks made on application level. These DoS attacks would be performed exploiting the authentication mechanism of REST calls, since every time a call is made, it is necessary to verify whether a token, generated at the beginning of the process, is valid or not. This might cause a system to overload if a huge amount of invalid tokens is sent during normal operation. Naturally, as our solution works only for DoS attacks on application level, it is important to include also some kind of protection to avoid DoS on network level. For example, if an attack exploits a TCP/IP failure or even a network package flood, then a solution like the one proposed by Tariq [19] would successfully protect the system. Therefore, if these, and other, solutions are used together, the system would be more resilient against DoS attacks.

It is important to notice that when our solution is running, there was an improvement on the response time when the system is under attack (see Fig. 9). The response time was improved in 36% when our solution is running, so even during a DoS attack, the system still responds in an acceptable time.

As future works we intend to analyze the attacks that happen during a time window. This time window will be used to verify the amount of valid and invalid requests that are being sent from the same IP address. This will be useful to avoid dropping packets from legitimate clients that are behind a NAT solution, *i.e.* if some attackers are executing behind a NAT, to hide invalid packets with valid packets, our extended solution would allow the CMS to reconfigure, dynamically, the percentage of packets from that IP that might be filtered or not.

Acknowledgment

The authors would like to thank Mauro Storch the discussions on cloud computing and to HP R&D Brazil for providing a scholarship through Brazilian law 8.248/91.

References

[1] L. D. Stein and J. N. Stewart, "WWW security faq: Securing against denial of service attacks.", 2003, http://www.w3.org/Security/Faq/wwwsf6.html (Access Date: 04 Jun, 2014).

[2] Feedly, "Denial of service attack [neutralized] — building feedly.", 2014, http://blog.feedly.com/2014/06/11/denial-ofservice-attack/ (Access Date: 05 Aug, 2014)

[3] W. Dawoud, I. Takouna, and C. Meinel, —Infrastructure as a service security: Challenges and solutions,‖ in 7th International Conference on Informatics and Systems (INFOS), 2010, pp. 1–8.

[4] D. Perez-Botero, J. Szefer, and R. B. Lee, —Characterizing hypervisor vulnerabilities in cloud computing servers,‖ in Proceedings of the 2013 international workshop on Security in cloud computing, 2013, pp. 3–10.

[5] I. VMWare, "Vmware virtualization for desktop server, application, public hybrid clouds.", 2014, http://www.vmware.com (Access Date: 03 Jul, 2014)

[6] I. Microsoft, "Virtualization for your modern datacenter and hybrid cloud.", 2014, http://www.microsoft.com/enus/server-cloud/solutions/virtualization.aspx (Access Date: 03 Jul, 2014)

[7] Xen, "The xen project, the powerful open source industry standard for virtualization.", 2014, http://www.xenproject.org/ (Access Date: 03 Jul, 2014)

[8] KVM, "Kernel based virtual machine.", 2014, http://www.linux-kvm.org/page/Main Page (Access Date: 03 Jul, 2014)

[9] P. M. e Timothy Grance, "The NIST definition of cloud computing.", 2002, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (Access Date: 23 Jul, 2014)

[10] OpenStack, "Openstack open source cloud computing software.", 2014, https://www.openstack.org/ (Access Date: 03 Sep, 2014)

[11] A. CloudStack, "Apache cloudstack: Open source cloud computing.", 2014, http://cloudstack.apache.org/ (Access Date: 23 Jul, 2014)

[12] Eucalyptus, "Eucalyptus open source private cloud software.", 2014, https://www.eucalyptus.com/ (Access Date: 23 Jul, 2014)

[13] Q. Lu, L. Zhu, L. Bass, X. Xu, Z. Li, and H. Wada, "Cloud API issues: an empirical study and impact," in Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, 2013, pp. 23–32.

[14] R. T. Fielding, "Fielding dissertation: Chapter 5: Representational state transfer (rest).", 2000, http://www.ics.uci.edu/ fielding/pubs/dissertation/restarchstyle.htm (Access Date: 03 Jun, 2014)

[15] R. Gracia-Tinedo, M. Sanchez Artigas, and P. Garcia Lopez, —Cloudas-a-Gift: Effectively exploiting personal cloud free accounts via REST APIs,‖ in Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, June 2013, pp. 621–628.

[16] OpenStack, "Welcome to keystone, the openstack identity service!", 2014, http://docs.openstack.org/developer/keystone/ (Access Date: 03 Jun, 2014)

[17] F. Kargl, J. Maier, and M. Weber, —Protecting web servers from distributed denial of service attacks,‖ in Proceedings of the 10th International Conference on World Wide Web, 2001, pp. 514–524.

[18] H. Beitollahi and G. Deconinck, "Tackling application-layer DDoS attacks" Procedia Computer Science, vol. 10, no. 0, pp. 432 – 441, 2012, ANT 2012 and MobiWIS 2012.

[19] U. Tariq, Y. Malik, B. Abdulrazak, and M. Hong, —Collaborative peer to peer defense mechanism for DDoS attacks,‖ Procedia Computer Science, vol. 5, pp. 157 – 164, 2011.

[20] J. Mirkovic and P. Reiher, —A taxonomy of DDoS attack and DDoS defense mechanisms,‖ SIGCOMM Computing Communications, vol. 34, no. 2, pp. 39–53, Apr. 2004.