PEP4Django - A Policy Enforcement Point for Python Web Applications

Carlos Eduardo da Silva[1], **Welkson Renny de Medeiros**[2], **Silvio Costa Sampaio**[1]

[1] Digital Metropolis Institute
Federal University of Rio Grande do Norte (UFRN)
Natal – RN – Brazil

[2]Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN)
Natal – RN – Brazil

{kaduardo,silviocs}@imd.ufrn.br, welkson.medeiros@ifrn.edu.br

Abstract. *Traditionally, access control mechanisms have been hard-coded into application components. Such approach is error-prone, mixing business logic with access control concerns, and affecting the flexibility of security policies, as is the case with IFRN SUAP Django-based system. The externalization of access control rules allows their decoupling from business logic, through the use of authorization servers where access control policies are stored and queried for computing access decisions. In this context, this paper presents an approach that allows a Django Web application to delegate access control decisions to an external authorization server. The approach has been integrated into an enterprise level system, which has been used for experimentation. The results obtained indicate a negligible overhead, while allowing the modification of access control policies without interrupting the system.*

1. Introduction

Access control has been traditionally employed to restrict access to resources in information systems [Sandhu and Samarati 1994]. Among the main mechanisms found in the literature, one of the most used is the *Role-Based Access Control* (RBAC) [ANSI 2004] model, which associates permissions to roles played by users in an organization.

This is the case of the SUAP (Public Administration Unified System - *Sistema Unificado de Administração Pública*) system targeted in this article and deployed at Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN). This system has been developed using the Python Django framework, and employs an access control mechanism based on RBAC. In its current implementation, The SUAP system employs the Django security mechanisms for access control, based on the *Django.auth* module, requiring the hard-code of access control rules into the business logic of application components. This is achieved by means of decorators, similar to Java annotations, on the methods to be protected, in which the necessary roles for accessing it are specified.

Such approach presents a restricted flexibility for security policies, as any change in a specific rule requires modifications on the source code and re-deployment of the system in production. Moreover, the developers of SUAP noticed the necessity of more complex access control rules, involving other information besides the user role. As a turnaround strategy, they have employed specific conditional statements in the source

code for checking access restrictions. In the course of time, however, this error-prone practice has spread over the different modules of the system, which combined with the RBAC role explosion problem [Elliott and Knight 2010] (with a huge number of roles currently employed by the system), reached a point where the management of access control rules is becoming a real problem.

In this paper, we introduce the PEP4Django approach as a solution to allow a Django web application to delegate access control decisions to an external authorization server, hence decoupling the access control management from the application code. Also, the improvements of our approach in terms of feasibility and performance are assessed through its deployment on the SUAP system. Results indicate a negligible overhead while allowing the modification of access control policies without interruptions to the system.

The remainder of this paper is structured as follows. First, we provide some necessary background to explain our approach in the section 2. In Section 3, the PEP4Django approach is detailed. The evaluation in Section 4 presents two experiments that was carried out to demonstrate the feasibility and performance of the proposed solution. Section 5 discusses related work. Finally, several key conclusions which have been drawn from this work are stated in section 6.

2. Background

To provide the necessary background to explain our approach, we overview some key concepts on access control as well as how they are implemented in the Django's security framework.

2.1. On Access Control

The mechanisms for implementing access control can be seen through a logical architecture commonly referred to as AAA-triad (Authentication, Authorization and Auditing)[1] [Sandhu and Samarati 1994]. Whenever a subject (or user) wants to access some resource, it first needs to authenticate itself, a procedure used to correctly establish the subject identity, confirming if the user is who he/she claims to be. Once authenticated, the subject needs to be authorized, which checks whether the subject holds the adequate privileges or permissions to perform the desired action on the specified resource. The auditing is concerned with the logging of information about the authentications and authorizations performed in the system, providing the means for a posteriori analysis of all activities of users in the system.

Amongst the different manners to implement access control mechanisms, we can mention the role based access control (RBAC) [ANSI 2004] and the attribute based access control (ABAC) [Hu et al. 2014] models. In the RBAC model permissions are associated to roles, which capture the different work functions inside organizations. Users are then assigned roles, effectively receiving the permissions that are associated to these roles. In the ABAC model permissions are associated with attributes, which can be represented as a key-value pair characterizing information about subject, object, operation or environmental conditions (such as date/time or IP address).

An access control mechanism follows a functional architecture, presented in Figure 1, in which its main components are identified. Whenever a subject tries to perform

[1] Some authors use the term accounting to refer to the same concept.

an operation in an object, the *Policy Enforcement Point* (PEP) intercepts the request and queries the *Policy Decision Point* (PDP) for a decision whether or not to allow access. The PDP recovers the relevant access control policy from a policy repository, and evaluate the policy based on the information received from the PEP. In case more information is needed to evaluate the policy, the PDP queries the *Policy Information Point* (PIP) for attributes from other sources. Once a decision is made, the PDP notifies the PEP, which then allows or denies the access accordingly. The *Policy Administration Point* (PAP) provides an interface for managing access control policies, which are stored in the policy repository.

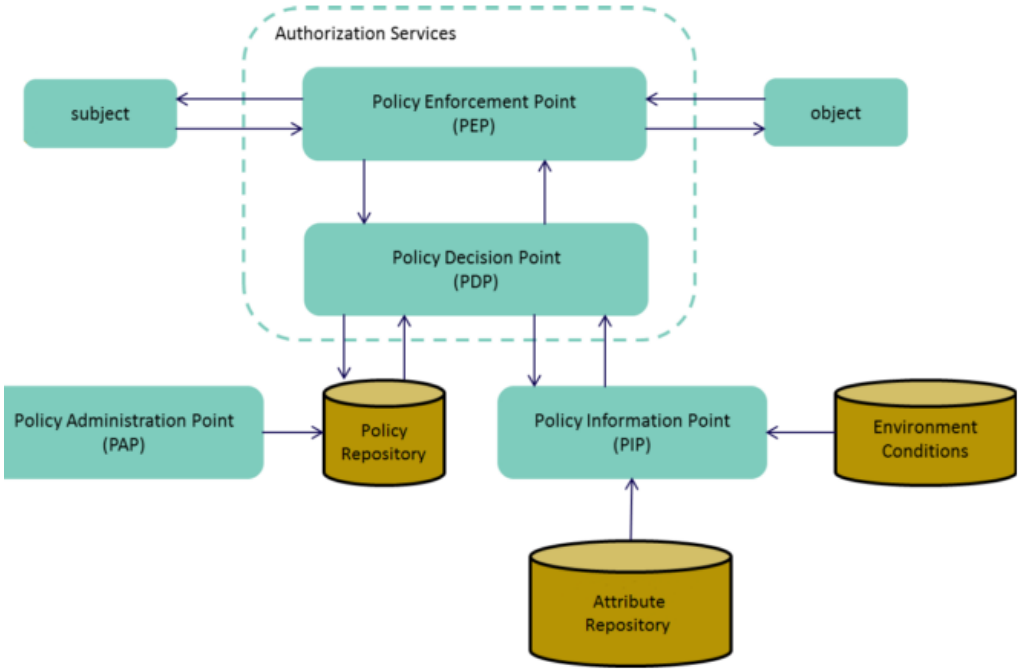

Figure 1. Functional architecture of access control mechanism [Hu et al. 2014].

2.2. Access Control in the Django Framework

Regarding authentication, the Django framework supports a variety of storage mechanisms and external authentication services. Conversely, regarding authorization, it is based on the use of groups, permissions and objects.

Django provides four default permissions (add, change, delete, view) that can be applied over object types, supporting the creating of personalized permissions. Groups are treated as attributes of users that must be maintained in the chosen user storage system. The association between groups and permissions can be made through a Web interface provided by the framework, or though API calls. However, the permissions are enforced by means of decorators in the source code, or through the use of if statements. Thus, in order to specify the necessity of a particular permission for performing an action, the programmer needs to explicitly include an annotation in its method implementation stating the necessary group or permission; or create several if statements checking whether the user has the desired permission. In case the access control rules need to use a particular attribute, it is necessary to create if statements checking the values of the desired attributes.

Such approach is not desirable in an ever growing and dynamic environment, in which Web system are becoming increasingly complex. Considering the SUAP system, in which this research has been applied to, we started to notice problems of role explosion, with an increasing number of groups being created and not curated appropriately. For example, people who left their old functions were not removed from the groups. Moreover, such static approach to access control was becoming a burden on the development team to accommodate changes in the access control policy.

3. The PEP4Django Approach

In this section, we present our approach and give the details on how to incorporate external access control into Python Django applications.

3.1. Integrating PEP4Django into the Django Architecture

Django is based on middlewares, in which an HTTP request is processed following a chain of responsibility design pattern, as shown in Figure 2. Our approach consists in creating a new Django middleware (*PEP4Django*)[2] that is then inserted into the end of Django processing chain. In this way, the different functions of Django security middleware are performed before the authorization request. For example, user-agents are dealt with by the *CommonMiddleware* component, and there are components for dealing with session (*SessionMiddleware*), protection against *Cross Site Request Forgery* (*CsrfViewMiddleware*), user authentication (*AuthenticationMiddleware*), message support based in sessions or cookies (*MessageMiddleware*), and other options.

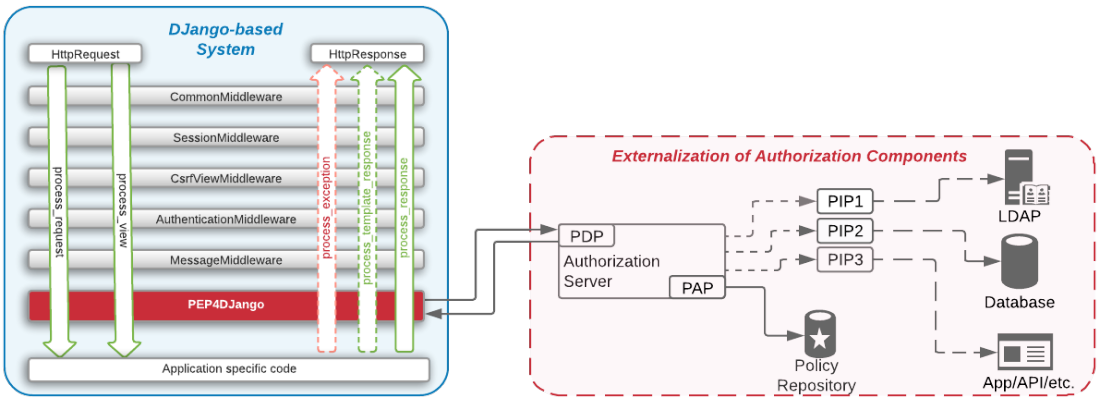

Figure 2. Architecture of Django framework integrated with PEP4Django for externalized authorization system.

The general inner workings of PEP4Django is based on interception of HTTP requests before they reach the respective business component (represented by *Application specific code*). Our middleware then collects information about the request, such as, the logged user (the subject in ABAC context), the URL of access (representing the resource), and the HTTP verb of the request (the operation). Based on the collected information, an authorization request is sent to the PDP in the external authorization service for access control decision, that are then enforced by PEP4Django.

[2]PEP4Django is available at `https://github.com/welkson/PEP4Django`.

In this way, all concerns related with access control policies are now managed in one single place, through the PAP component of the authorization service. Moreover, in case there is the need for a PIP, the same can be integrated into the authorization service. For example, Figure 2 shows PIPs for an LDAP directory service, a Database service, and external applications or APIs.

3.2. Implementation

PEP4Django has been implemented exploring the eXtensible Access Control Markup Language (XACML) [Parducci et al. 2013]. XACML has been proposed by OASIS, and can be considered the *de-facto* standard for specification of ABAC access control policies. It defines several elements for policy definition, and for request/response messages exchanged between PEP and PDP. Besides that, XACML has been supported by a number of commercial and open-source authorization services.

In our solution, we have employed the WSO2 Identity Server (WSO2 IS)[3] as authorization server. It can act as authorization service, providing REST API for the PDP and SOAP endpoint for the PAP, supporting both RBAC and ABAC models. It also provides modules for working as service provider (SP) and identity provider (IdP), supporting single-sign-on through OpenID Connect, SAML 2.0 and WS-Federation. The WSO2 IS also provides a Web-based interface for the definition of XACML access control policies, and to evaluate policies by simulating authorization requests.

Thus, PEP4Django sends an XACML request through WSO2 IS Rest interface, receiving an XACML response with the access control decision. In case the response is *Permit*, the HTTP request is forwarded to the Application specific code. In case of a *deny* or any other response the HTTP request is denied with a response code 403 Forbidden.

3.3. Integrating PEP4Django into an Running Environment

In order to use PEP4Django in your Web application it is necessary to include an entry into the Django configuration file. The entry is added to the "MIDDLEWARE" configuration group, as shown in Listing 1, in the last line, indicating that it is the last middleware in the chain.

```
MIDDLEWARE = [
  'django.middleware.common.CommonMiddleware',
  'django.contrib.sessions.middleware.SessionMiddleware',
  'django.middleware.csrf.CsrfViewMiddleware',
  'django.contrib.auth.middleware.AuthenticationMiddleware',
  'django.contrib.messages.middleware.MessageMiddleware',
  'Demo.middleware.django_pep.DjangoPEPMiddleware',
]
```

Listing 1. Activation of PEP4Django in the middleware group of settings.py file.

It is still necessary to define some configuration parameters specific of PEP4Django in the Django configuration file. The parameters are described in the sequence: The parameter `DJANGOPEP_PRODUCT` indicates the authorization server to be used. Our current implementation supports WSO2-IS (value "1") and Fiware's AuthZ-Force (value "2"). Parameter `DJANGOPEP_URL` indicates the URL of the authorization

[3]https://wso2.com/identity-and-access-management

server's PDP interface. Parameters DJANGOPEP_USER, DJANGOPEP_PASSWORD and DJANGOPEP_TOKEN are used for authenticating PEP4Django in the authorization server. We support authentication with username and password, or through OAuth tokens. Parameter DJANGOPEP_DEBUG enables the debug mode, in which the XACML Request and Response are logged. The parameter DJANGOPEP_IGNORE is used to specify a list of URLs to be ignored by PEP4Django. For example, the */accounts/login/* URL is used as login screen by the Django framework, and thus should be ignored by our PEP.

An example of PEP4Django configuration is presented in Listing 2. In this example PEP4Django is configured to operate with WSO2-IS, running at the same host as the Django application server (localhost) at port 9443. The remain of the URL (/api/identity/entitlement/decision/pdp) is the default PDP interface of WSO2-IS. We have used standard username/password authentication at WSO2-IS (admin user) and activated debug. We have also included some URLs that should be ignored by PEP4Django, such as "/admin/*" and "/accounts/*", which provide default services by the Django framework. The "/comum/*" and "/media/*" URLs contains static files that should not be protected in our example.

```
# PEP4Django Middleware settings
DJANGOPEP_PRODUCT = 1 #
DJANGOPEP_URL = 'https://localhost:9443/api/identity/entitlement
      /decision/pdp'
DJANGOPEP_USER = 'admin'
DJANGOPEP_PASSWORD = 'example'
DJANGOPEP_TOKEN = ''
DJANGOPEP_DEBUG = True
DJANGOPEP_IGNORE = ['/$', '/admin/*', '/accounts/*', '/comum/*',
      '/media/*']
```

Listing 2. PEP4Django middleware configuration parameters in file settings.py.

4. Evaluation

In this section we present some of the experiments that have been performed for evaluating PEP4Django. We initially describe the experiment performed for evaluating the correct implementation of the access control policy. In the sequence, we present some initial performance experiments to evaluate the impact of our solution in the system.

4.1. Business Process and Access Control Policies Description

In order to evaluate the correct working of our proposed solution, we have considered one of the business processes of the SUAP system. Figure 3 presents the business process for the support ticket service offered by the system using UML activity diagrams. This service contains an access control policy that has been defined based on the RBAC model, with three different roles: *Client*, *Support*, and *Admin*. The *Client* represents any user that needs IT support, which can be registered by opening a ticket. *Support* attendants are members of the IT team, responsible for handling the tickets raised by clients. *Admin* supervises the work of the IT support team, interacting with the service when necessary.

This service includes permissions to open tickets, either by a client (*1. Open Ticket*) or by a support attendant on behalf of a client (*2. Open ticket of behalf*), e.g., a client went in person to the IT department to report a problem. A client can cancel

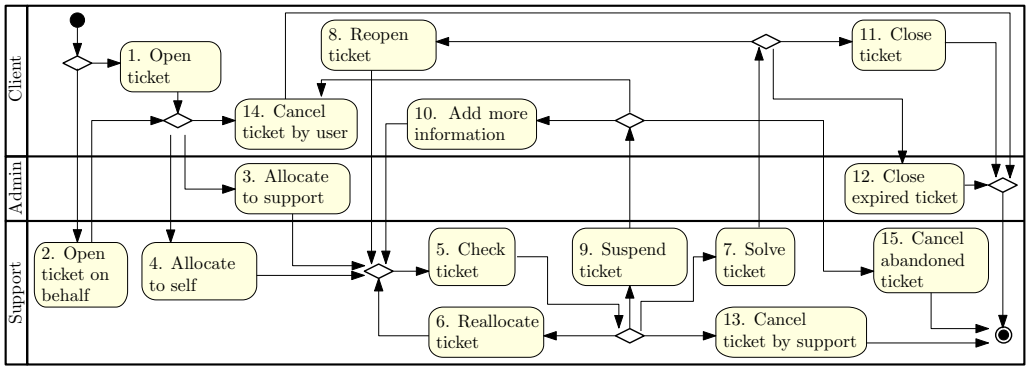

Figure 3. Example of business process for the ticket support service of system XYZ and its associated roles.

an open ticket before it has been allocated (*14. Cancel ticket by user*). Open tickets are allocated to support attendants, either by themselves (*4. Allocate to self*) or by an administrator (*3. Allocate to support*). Once allocated a ticket is handled by a support attendant (*5. Check ticket*), which can reallocate the ticket to another support attendant (*6. Reallocate ticket*), mark it as solved (*7. Solve ticket*), cancel the ticket (*13. Cancel ticket by support*), or suspend the ticket asking for more information from the client (*9. Suspend ticket*). Suspended tickets are returned to the client to include more information about the issue (*10. Add more information*). Clients can also cancel the ticket at this point (*14. Cancel ticket by user*). Once a ticket is marked as solved, the client can then close ticket (*11. Close ticket*) indicating that the issue has been resolved, or reopen the ticket (*8. Reopen ticket*), which is then reallocated to the same support attendant who dealt with it. Solved tickets that are not closed or reopened by clients can be closed by administrators (*12. Close expired ticket*). Similarly, suspended tickets are considered as abandoned when the user does not interact with it after an specific amount of time and can be canceled by support attendants (*15. Cancel abandoned ticket*).

The SUAP system uses a service-oriented architecture, in which the support ticket business process is exposed as a RESTfull API. For example, the opening of a new ticket consists in a POST call to the resource "`/new_ticket`". Based on this, we have captured the access control policies ot the support ticket service as an XACML policy.

```
policy new_ticket {
  apply denyUnlessPermit
  rule can_new_ticket {
    target clause ticket.resourceId == "/new_ticket"
         clause ticket.actionId == "POST"
         clause user.role == "client" or
              user.role == "support"
    permit
  }
}
```

Listing 3. XACML policy for "create new ticket" operation expressed in ALFA.

Listing 3 presents an excerpt from the access control policy of the support ticket service for the *Open new ticket* permission. This policy has been defined in XACML

using the ALFA[4] Eclipse plugin, which provides a simplified syntax for writing policies that are converted into XACML, that are then inserted into WSO2-IS PAP interface. This policy specifies the attributes expected for the resource (`ticket.resourceId == "/new_ticket"`), the operation (`ticket.actionId == "POST"`), and the expected roles (`user.role == "client" or user.role == "support"`). This policy states that to open a new ticket the user must have a role of client or support.

4.2. Environment

We have created three different environments to evaluate our approach. The first environment, which we call Simulated-SUAP, is based on a REST service that duplicates the support ticket service API, returning a set of standard responses to the received requests. This service has then been protected with PEP4Django. The other two environments have been created to evaluate the impact of introducing PEP4Django and external authorization into an existing system. These consists on a copy of SUAP system, which is called Legacy-SUAP, and a fork of SUAP (Externalized-SUAP) in which the authorization mechanism for one of its modules has been replaced by PEP4Django. This involved the removal of the authorization related code, the definition of XACML policy that captures the correspondent access control rules, and the deployment of the new system in a duplicate environment.

Legacy-SUAP and Externalized-SUAP systems can operate over the same database system, which has been replicated from the production environment. Both systems have been deployed in separate virtual machines, alongside a third virtual machine for the database server, all with the same configuration and amount of resources of the production environment: 16 CPU cores and 32GB of RAM. Besides that, we have deployed the authorization server on a separate virtual machine (8 CPU cores and 12GB of RAM), and created another virtual machine for acting as load generator (32 CPU core and 32 GB of RAM) using the JMeter [5] software.

4.3. Feasibility Experiments

In order to demonstrate the feasibility of our approach, we have conducted two sets of experiments. These experiments were based on the test cases used by the system developers regarding the support ticket service. We used those test cases that considered access control restrictions.

The first set of experiments involved the use of an HTTP and XACML request generator developed by us, together with Simulated-SUAP environment. Our request generator incorporates PEP4Django, and can be used to simulate user access to the support ticket service. It creates HTTP requests to the protected resources based on a set of parameters of a configuration file, and uses the HTTP request to create an XACML request that is sent to the PDP interface of WSO2-IS. These initial experiments have been used to assess the XACML requests produced by our code against the PDP interface, and to evaluate the access control policy saved on WSO2-IS. Based on these experiments, we have been able to identify and correct some bugs in PEP4Django regarding the creation of XACML requests based on HTTP requests.

[4]https://www.axiomatics.com/product/developer-tools-and-apis/

[5]https://jmeter.apache.org/

The second set of experiments used the Externalized-SUAP deployment. We started by doing manual tests with some scenarios of the support ticket service, and then moved to use the actual test cases of the system. In this experiment, the user attributes (i.e., userId and role) have been obtained from the HTTP session, the strategy normally adopted by SUAP developers. Thus, PEP4Django included the attributes in the XACML request sent to the PDP. We have then created some new users in the system, allocating different roles, such as "visitor", "client", "support" and "admin".

Figure 4 presents an example where an user with the *visitor* role tries to create a new ticket (represented by the *POST* action over the resource */new_ticket*). In this case, the policy states that only users with the *support* role are authorized to create new tickets, and the PDP denies the access because of the mismatch between the user role and the required role.

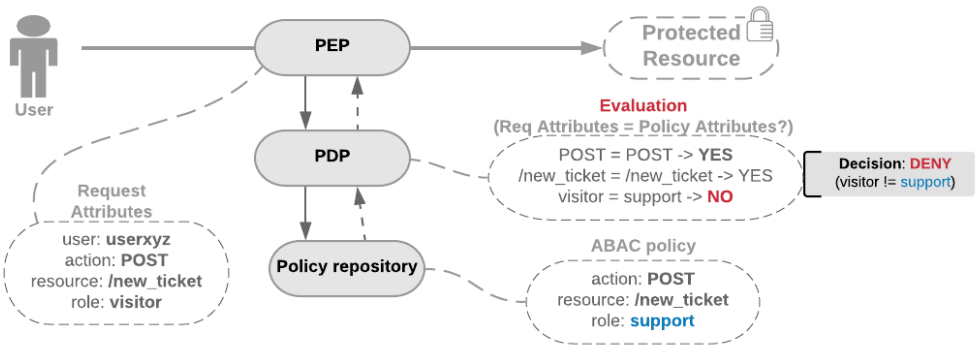

Figure 4. Example of ABAC-based authorization decision mechanism.

We repeated this experiment varying the user and the role attribute. The PDP was able to correctly allow or block access to users based on the attributes passed in the XACML request. We have also experimented with changes to the access control policy, using the PAP Web interface of WSO2-IS. There were no changes perceived by the final user, and we were able to modify the access control policy without changes in the source code of the application.

4.4. Performance Experiment

In the sequence, we have performed some initial experiments to evaluate the impact of PEP4Django in the overall system performance. The use of an external authorization mechanism introduce an additional step in serving user requests, and we were interested in verifying whether PEP4Django would significantly increase the system response time for the final user.

The performance experiments involved the definition of test plans for JMeter in a desktop computer, which were then moved to the load generator virtual machine, where a headless JMeter has been started, according to JMeter best practices[6]. In these initial experiments, the test plan simulates 100 users making requests to the system, and it has been repeated 10 times in each of the systems (Legacy-SUAP and Externalized-SUAP). This number of users has been chosen as it represents the average number of users that

[6]https://jmeter.apache.org/usermanual/best-practices.html

simultaneously use the support ticket service. This value has been found based on the monitoring and analysis tools employed by SUAP administrators.

For this particular experiment, JMeter generates requests to random tickets, in a database with over 72 thousand tickets. This has been done to diminish the impact of cache on the several components of the infrastructure. The average time for the Legacy system is around 4854 milliseconds, while the average time for the Externalized system is around 5088 milliseconds. These are very promising results, showing that our approach with externalized access control mechanisms produces an overhead of 4.81% over the Legacy system, with an absolute difference of 233 milliseconds.

4.5. Discussion

Although initial, the obtained results are very encouraging and indicates the feasibility of our approach. The experiments performed have shown that PEP4Django can be used as means for externalization of access control in Django applications. In particular, we draw attention to the capability of changing access control policies without touching the source code of the application, as this was a manual and error-prone process that requires the re-deployment of the system. The initial performance experiments have shown an overhead of only 233 miliseconds (4.81%) for 100 simultaneous user. However, more performance related experiments are needed to show, for example, the curve of our solution with the growing number of users.

5. Related Work

An extensive survey conducted by Servos and Osborn [Servos and Osborn 2017] have shown that ABAC has been received increasing attention over the years, with the vast majority of works focusing on the definition of ABAC models. There is also a considerable amount of work on the definition of access control policies, and several studies employ XACML as policy language, which has been used in several commercial products for access control.

In order to implement ABAC access control mechanisms in an architecture with externalized functional components, we chose to deploy an authorization server with support to the XACML language [Brossard et al. 2017]. There are several options available, such as the ACROSS framework [Silva et al. 2018], and FIware AuthZForce[7]. However, ACROSS provides a scope far beyond what we need, and AuthZForce has a very high coupling with other FIware identity management components. Moreover, we consider both approaches to be in an experimental stage, and not ready for a production environment. Whit this is mind, we looked at *AT&T XACML*[8] reference implementation. It has been developed in Java and provides REST API for its PAP and PDP interface, and a Web interface for managing policies. However, we considered it as not ready for a production environment as it requires some coding effort for its deployment and integration into a running system. We have also looked into OpenAM[9]. Although it offers support for SAML, OAuth, and OpenID Connect, besides XACML, it is now only offered as a commercial product called *ForgeRock Identity Management*[10].

[7] `https://authzforce-ce-fiware.readthedocs.io`

[8] https://github.com/att/XACML

[9] https://github.com/ForgeRock/openam-community-edition

[10] https://www.forgerock.com/platform/identity-management

We decided for WSO2 Identity server based on its active open-source community, support to functionalities not available in the other options and the possibility of hiring a company in case of an enterprise deployment. WSO2-IS provides a REST API for its PDP interface, and a SOAP API for its PAP interface. It also supports single-sign-on with different protocols (*OpenID Connect*, *SAML 2.0* and *WS-Federation*), supporting its use as both Identity Provider and Service Provider. Its Web interface provides assistants for the definition of access control policies, and support the simulation of policies by allowing the definition of XACML requests.

Some works have focused on externalized access control mechanisms in different contexts. Sette et al. [Sette et al. 2017] presented a similar approach, where externally defined and stored policies are translated into platform specific rules, but including support to identity federation for authentication. Domenech et al [Domenech et al. 2016] have proposed an approach for controlling access on the Internet of Things, with mechanisms for considering access control rules enforcement on constrained devices. All these approaches demonstrate the use of externalized access control mechanisms, but specific to their respective application domains. In fact, the work closest to ours has been presented by [Armando et al. 2014], in which an ABAC enforcement component has been developed for the Spring Framework. Their approach is based on the use of dependency injection and the decorator pattern, in which a Java annotation is included at the method to be protected. In this way, they still require source code alteration for protecting resources. Different from their approach, our solution works at the API level, leveraging the REST architectural style and HTTP request interceptions, without requiring changes to source code whatsoever.

Although there are several works on the topic of implementing access control mechanisms, we have not found many works on the actual integration of ABAC access control into existing systems. We took inspiration for our approach from [Brossard et al. 2017], which presents an approach for implementation ABAC access control in companies. Their approach is based on a life-cycle for authorization policies composed by eight phases, from use case definition and authorization requirements elicitation, to definition, testing and deployment of policies on the target system. Although they provide some guidance on the implementation of ABAC policies, their approach is too abstract, in contrast with ours, that has been applied to a real system.

6. Conclusion

This paper presented PEP4Django, a policy enforcement point for externalizing access control management and handling for Django Web applications. The approach has been implemented as a Django middleware, without requiring any changes in the original Django source code. In this way, we also do not require any changes in the application source code for dealing with access restrictions, nor to change access control policies. The approach is based on the XACML language and the WSO2 Identity Server authorization server, but could be easily modified for supporting other existing products.

As previously mentioned, we have applied PEP4Django in the context of the SUAP system, a real system that provides services to more than 35,000 users of IFRN in Brazil. In order to evaluate our approach, we have modified the SUAP system for using externalized access control mechanisms in one of its modules. The obtained results

shown an overhead of 4.81% when introducing PEP4Django for externalizing access control in the SUAP system, with an absolute overhead of around 233 milliseconds for 100 simultaneous users. This has been considered an excellent result, as the experimental environment did not consider replication nor load-balancing techniques employed in the production environment.

Although we obtained very encouraging results, some limitations need to be considered. It is necessary to run a more comprehensive performance experiment, with varying load in order to study the overhead of the solution over a growing number of users. Also, it is necessary to perform some experiments involving attributes than those included in the HTTP request received by PEP4Django.

References

ANSI (2004). Role Based Access Control. Technical Report ANSI INCITS 359-2004, ANSI. ANSI/INCITS 359-2004.

Armando, A., Carbone, R., Chekole, E. G., and Ranise, S. (2014). Attribute based access control for apis in spring security. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, pages 85–88. ACM.

Brossard, D., Gebel, G., and Berg, M. (2017). A Systematic Approach to Implementing ABAC. In *Proceedings of the 2Nd ACM Workshop on Attribute-Based Access Control*, ABAC '17, pages 53–59, New York, NY, USA. ACM.

Domenech, M. C., Boukerche, A., and Wangham, M. S. (2016). An authentication and authorization infrastructure for the web of things. In *Proc. of the 12th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, Q2SWinet '16, pages 39–46.

Elliott, A. and Knight, S. (2010). Role Explosion: Acknowledging the Problem. In *Software Engineering Research and Practice*, pages 349–355.

Hu, V. C., Ferraiolo, D., Kuhn, R., Friedman, A. R., Lang, A. J., Cogdell, M. M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al. (2014). Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST special publication*, 800(162).

Parducci, B., Lockhart, H., and Rissanen, E. (2013). Extensible access control markup language (XACML) version 3.0. *OASIS Standard*, pages 1–154.

Sandhu, R. and Samarati, P. (1994). Access Control: Principles and Practice, IEEE Communications 32 (9): 40–48, 1994. *IEEE Communications Magazine*, 32(9):40–48.

Servos, D. and Osborn, S. L. (2017). Current Research and Open Problems in Attribute-Based Access Control. *ACM Computing Surveys*, 49(4):65:1–65:45.

Sette, I. S., Chadwick, D. W., and Ferraz, C. A. G. (2017). Authorization policy federation in heterogeneous multicloud environments. *IEEE Cloud Computing*, 4(4):38–47.

Silva, E. F., Muchaluat-Saade, D. C., and Fernandes, N. C. (2018). ACROSS: A generic framework for attribute-based access control with distributed policies for virtual organizations. *Future Generation Computer Systems*, 78:1–17.