ZeroVM enabled Content Based Access Control for Swift Storage

Prosunjit Biswas
University of Texas At San antonio
Email: prosun.csedu@gmail.com

Farhan Fatwa
University of Texas at San Antonio
Email: Farhan.Patwa@utsa.edu

Ravi Sandhu
University of Texas at San Antonio
Email: ravi.sandhu@utsa.edu

***Abstract*—While Openstack swift facilitates storage and management of unlimited amount of data, with conventional cloud computing paradigm, for the purpose of processing these enormous amount of data is required to be moved back and forth to the computing host exhausing significant cpu cycle and resulting serious I/O bottleneck. ZeroVM, a specially designed hypervisor for the cloud, eliminates unnecessary movement of data by enabling data local computing by virtue of specially designed applications called ZAP (zerovm application package) which facilitates computation around data inside swift storage. This approach can enables swift customers to have sophisticated control over their data by specifying not only who can or cannot access their data, but also how much of the data can be accessed. Inspired by this fact, we are developing a zerovm application, that let data ownser specify access policy on the content of the data file and describe who should or should not access which portion of the data which is essentially more fine granined access control over the ACL based all/nothing access. The prototype of our proposal is implemented on JSON data formatted file.**

I. Introduction

OpenStack Swift is a highly deployed opensource cloud storage solution. With its unlimited storage capability, it is used to store any number of large / small objects through its RESTful HTTP API. A user can submit a GET request to download a file and a PUT request to upload a file. But a fundamental problem in the cloud storage system is that whenever data is required to be processed, it has to be moved to the computing hosts (VM, EC2 instance) before computation and moved back to the original storage afterward which results significant I/O overhead.

ZeroVM [4], an specially built hypervisor for the cloud promises to solve the problem of secure computation. This is an application vertualization technique based on google native client (NaCl) project [5] which is able to run arbitrary (and potentially malicious) code and still provide security guarantee. Unlike existing solution like docker [11] which is also very exciting in its own merit, zerovm focuses more fault isolation and secure computation.

This new technoloy whenever integrated with swift storage, is able execute arbitrary application (and potententially unsafe code) inside swift cluster and process data locally. With its tight security guarante, Zerovm assure both the data owner and storage provider from potential security risks from completely untrusted application enabling data local in storage computation. This new paradigm introduces whole lot of opportunities. For example, now it is possible to search data while in storage, look for patterns, or serve a file partially along with exciting use case like running query for big data and extracting salient customer pattern or product demand.

The integration of these two new technologies also open up an exciting era for both cloud storage provider and cloud customer. From the perspective of storage provider, along with the storage they can also offer useful data processing application which may help customer to get better service and even save money which was spent due to the movement of data. From the customers perspective, they no longer need to provision large cluster that they used to use for the processing of the data.

As an effort to develop a zerovm application, we are proposing content based access control for object/files stored in the object store. we would enable swift customers to specify who can access how much content of their data. To give a concrete example, consider a hospital stores its patient record in the object store. Now, the record files should be accessed differently by different personnels. For example, the doctor can see certain part while the billing accoutant should see other part of the record. Our application would let the data owner specify policy expressing who can see which part of the data.

As a prototype implementation, we would work with JSON formatted file because of several reasons. Firstly, JSON is gaining immense popularity due to its concise representation and easiness in human and machine readability. Secondly, industries are increasingly adopting JSON for internal data representation and data exchange format which is reflected by the facts that JSON document database such as MongoDB(more accurately BSON, a modified version JSON) is now officially supported by the OpenStack cloud platform; twitter latest api (v 1.1) supports only JSON and youtubes latest API (v 3) [9] recommend JSON as the default exchange format. Thirdly, we believe that JSON could be a easily adapted for semi-structured / unstructured big data.

II. Background

To keep the readers comfortable to our work, we would introduce the concepts of Attribute Based Access Control (ABAC) model very briefly.

A. Attribute Based Access Control

Attribute-based access control defines a new access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together. The policies can use any type of attributes (user attributes, resource attribute, etc) [8]

The advantage of using ABAC model is that attributea are very natural way of representing properties of users or objects. New attributes values or even new attributes can be easily added to the model. ABAC policy is also very flexible and expressive enough to configure most of the real world scenarious. Figure 1 is the abac model that our work is based on.

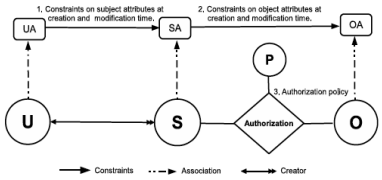

Fig. 1. Attribute Based Access Control Model.

As shown in the diagram, the authorization policy in this model (shown by the diamond in the figure) is specified with attributes from the subject/users and objects. Without giving any details, we would like to mention that the policy can include any number of user or object attributes.

The interested reader are encouraged to have a look at [1] where the model components and policy language is discussed in great detail.

III. Proposal

With Swift, the available access control mechanism is ACL (access Control List) which specify who can or cannot access an object or a file. In this approach, if someone can access a document, he/she gets the full content of the document which is an all or nothing approach. Figure 2 and 3 illustrates the situation where in the former case, a file is accessed in all / nothing approach and in the later case the file can be selectively accessed by different users. But we believe that zerovm enables more sophisticated cases which require more flexible access control than the ACL. For example, a hospital that stores patient medical record in the cloud, wants all its doctor, nurse or patient access the partial content based on the available role of the requester. So, along with the content, the owner (in this case, the hospital) of an object/file may need to specify who can access how much of the content.

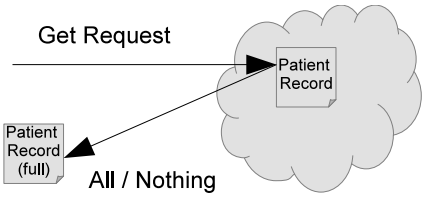

Fig. 2. Accessing file with swift API.

Figure 4, shows a sample file to be stored in the object store. For our example, we specify following different user roles namely, doctors, patient, nurse and billing stuff. A sample policy to be specified by the content owner is shown below.

1) Patient own 'personal_Information' and 'physicalExam' records. (s)he can read them.

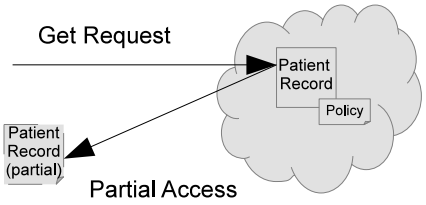

Fig. 3. Our Proposed Solution where file can be accessed selectively

Fig. 4. Labeled json data.

2) Patient allow doctors to read her 'physicalExam' records .
3) Doctors can read the entire medical records except information owned by the patient.
4) Nurses can read objects identified by 'health_record'.
5) The billing stuffs can only read 'Identification' information.

In order to formulate these policy, we would use ABAC (Attribute Based Access Control) model [1]. In ABAC, user, object is associated with attribute sand these attributes are used to specify policies. In [1], the authors have provided a simple and easy policy language which is expressive enough to capture Popular Access control models like DAC(Discretionary Access Control) [2], RBAC (Role Based Access Control) [3]. To be able to configure DAC and RBAC is important in the sense that the first policy in the above mentioned policies is a DAC policy and the rest are RBAC policies.

In order to specify these policies, we are developing a theoritical work for Access Control model for JSON data where we require user attributes like user role and object attributes like owner and object-label but for the shake of brevity, we are not representing details of the JSON Access Control model here. Worth to mention that in order to capture user-role we are exploiting the group feature of Identity API version 3 [9]

Now, whenever a user having role doctor request to get the whole file, he would be able to access only the content as

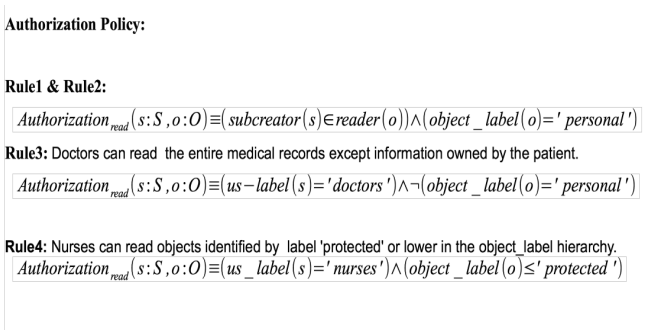

Fig. 5. Configured ABAC policy for given policy.

specified in listing 1 . Figure 5 shows the configured ABAC policy used in our implementation.

```
{
 "medical_record": {
      "physical_exam": {
    "appearance": "well developed",
    "eyes": "conjunctiva"
  },
  "Medications": [
            "PRINIVIL TABS 20 MG ",
            "Last Refill: #30 x 2 "
        ]
}
}
```

Listing 1: Content of Medical Record Object as Accessed by a User Having Doctor Role

Again, we envision that it should be possible to request the file by specifying a jsonpath along with the filename. For example, a requester having role 'doctor' should be able to access only medication information by specifying a json path argument ("//medication") along with the request line. A hypothetical command for the above query would be

Swift download container patient_record.json –jsonpath="//medication"

To sum up our proposal of Content Based Access Control, we want to achieve following:

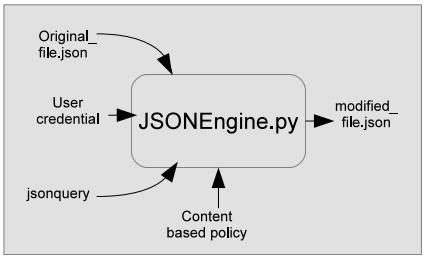

Fig. 6. The Skeleton for the CBAC zap.

1) Attach policy with a file/object stored in swift storage. The file can be requested as it it, or it can be partially requested by specifying query parameter(jsonpath in our case) and instead of having the full content, the requester may get selective content based on his acting role. This case is explained in the above section. The skeleton for the to be developed zerovm application is shown in the figure 6

IV. Implementation

Our implementation does not require any change in the zerovm project. We only require following changes in zerocloud middleware and zerovm client program

1) Changes in the swift request header
2) Changes in the zerocloud middleware
3) Changes in the swift client

A. Request header

In order to associate policy with the file and specify jsonpath as a query parameter, we need to add two request headers namely, **–cbac-policy** and **–jsonpath**. The values of –cbac-policy header can be a separte policy file or json text specifying the policies. On the otherhand, the value of –jsonpath would be a valid json path.

B. Zerocloud Middleware

In the zerocloud middleware, along with the file to be queried we need to capture the values of –cbac-policy (or retrieve corresponding policy file). We may need to modify the zerovm manifest file to include this policy file as another valid input channel.

C. zerovm client

In order to specify –cbac-policy and –jsonpath headers, we may need to modify python-swiftclient or zpm command.

V. Conclusion

As more and more data is being uploaded in the cloud, these data may contain sensitive information. With existing swift API, one can either access the full content or nothing. We have presented egitimate situations where one should be given access to a file with sensitive content being filtered out. With the coupling of swift with zerovm, we are proposing here to develop an application where someone can specify policy with a file and let different user access different parts of it. Currenly, our solution (content based access control) is limited to one document, but it the future we want to implement it for multiple linked document.

Acknowledgment

The authors would like to thank RackSpace, the open cloud company for supporting this project.

References

[1] Xin Jin, Ram Krishnan, Ravi Sandhu *A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC* 2012.

[2] Sandhu, Ravi S., and Pierangela Samarati. *Access control: principle and practice.* Communications Magazine, IEEE 32.9 (1994): 40-48.

[3] Sandhu, Ravi S., et al. *Role-based access control models.* Computer 29.2 (1996): 38-47.

[4] *zerovm* available at : http://zerovm.org/

[5] *Google Native Client* available at : json.org

[6] *JSON* available at : https://code.google.com/p/nativeclient/

[7] *mongodbt* available at : http://docs.mongodb.org/manual/

[8] *Google Native Client* available at : http://en.wikipedia.org/wiki/Attribute_Based_Access_Control

[9] *OpenStack Identity API V3* available at : http://developer.openstack.org/api-ref-identity-v3.html

[10] *ZeroVM Package manager* available at : http://zerovm-zpm.readthedocs.org/en/latest/

[11] *Docker* available at : http://www.docker.com/

[12] *mongodbt* available at : http://docs.mongodb.org/manual/