ZeroVM enabled Content Based Access Control for Swift Storage

Prosunjit Biswas Farhan Fatwa Dr. Ravi Sandhu
Computer Science Department and Cloud and Big Data Laboratory
The University of Texas at San Antonio
eft434@my.utsa.edu farhan.patwa@utsa.edu 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 and thus exhausing significant CPU cycles 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 uniquely designed applications called ZAP (ZeroVM Application Package) which facilitates computation around data inside Swift storage. This approach can enable Swift customers to have sophisticated control over their data by specifying not only who can or cannot access the data, but also how much of the content can be accessed. Inspired by the fact, we are developing a ZeroVM application that let data owner specify access control policy on the content of the data file and describe who can or cannot access which portion of the file which is essentially a more fine granined approach over the ACL based all/nothing access. The prototype of our proposal will be implemented on JSON data formatted file.**

I. Introduction

OpenStack Swift is a highly deployed open source 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 (ex. 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 on 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 data is in storage, look for patterns, or serve a file partially along with other exciting use cases 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 applications which may help customers 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 clusters 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 the data. To give a concrete example, consider a hospital stores its patient record in the object store. Now, these 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 attributes are very natural way of representing properties of users or objects. New attribute 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.

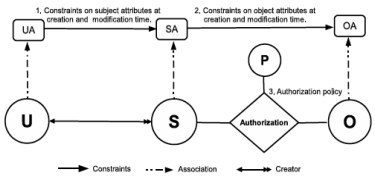

Fig. 1. Attribute Based Access Control Model.

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

III. Proposal

With Swift, the available access control mechanism is ACL (Access Control List) which specifies 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-or-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 records in the cloud, wants all its doctor, nurse or patient access selective 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.

Figure 4, shows a sample file to be stored in the object store. For our example, we specify following different user roles namely, doctor, 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 (see Fig. 4). Only the owner can read it.
2) Patient allow doctor to read her 'physicalExam' records .

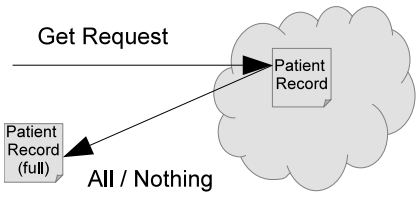

Fig. 2. Accessing file with Swift API.

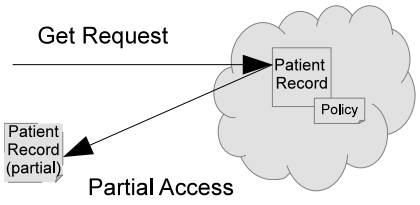

Fig. 3. Our Proposed Solution where file can be accessed selectively

Fig. 4. Labeled JSON data.

3) Doctor can read the entire medical records except information owned by the patient.
4) Nurse 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 attributes and 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]

Authorization Policy:

Rule1 & Rule2:

$$Authorization_{read}(s:S, o:O) \equiv (subcreator(s) \in reader(o)) \wedge (object\_label(o) = 'personal')$$

Rule3: Doctors can read the entire medical records except information owned by the patient.

$$Authorization_{read}(s:S, o:O) \equiv (us-label(s) = 'doctors') \wedge \neg(object\_label(o) = 'personal')$$

Rule4: Nurses can read objects identified by label 'protected' or lower in the object_label hierarchy.

$$Authorization_{read}(s:S, o:O) \equiv (us\_label(s) = 'nurses') \wedge (object\_label(o) \leq 'protected')$$

Fig. 5. Configured ABAC policy for given policy.

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

1) Attach policies 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

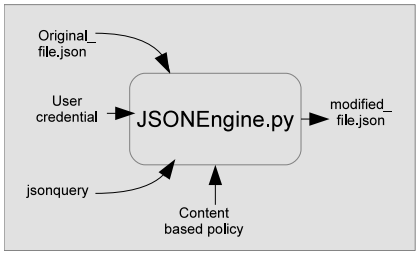

Fig. 6. The Skeleton for the CBAC zap.

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 JSONPath.

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. Program Schedule

The timeline of the deliverables of our project can be broken into small pieces to make it more concrete and achievable. Here we like to mention following phases of our project.

A. Extended Proposal,

After the submission of our initial proposal, here we are submitting an extended proposal that specifies in more details what we want to accomplish.

B. Design Phase

In this phase (by the end of August), we expect to have a detailed design of our implementation along with a concrete definition of the problem and the solution. We hope, during this phase, we should be able to evaluate our proposed approach along with its pros and cons.

C. Implementation Phase

In this phase (by the end of February, 2015), we expect to be able to implement our proposed solution, have an idea how to take it to the production line (if possible).We should also be able to drive business use cases for Rackspace customers for our proposed application.

D. Final Phase

In this phase (in the 2nd year), we would work on the enhancement of our prototype implementation. Currently, our proposal is Content Based Access Control for single document. During this time, we would work for multiple documents which seems more interesting and would probably meet real customer requirements.

VI. Conclusion

As more and more data is being uploaded in the cloud, data may contain sensitive information. With existing Swift API, one can either access the full content of a file or nothing. We have presented a legitimate situation 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 a policy with a file and let different user access different parts of it. Currenly, our solution which is a content based access control, is limited to one document, but it the future we want to implement it for multiple linked documents.

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] *mongodb* 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/

APPENDIX A
DESIGN DOCUMENT

A. Content Based Access Control for Swift Storage

Our proof of concept prototype works only on JSON formatted data stored in swift.

1) Usecase: Alice has a data file ('data.json') stored in Swift. With Swift-ACL she can specify who can or cannot access the file, but she also want to define who can access how much content of the file. In other words, besides using ACL, she wants to specify access control on the content level of the file. So, she attaches a policy ('policy.json') with the file and wants that all the access request to the file would honor the policy associated with the file.

2) Design Details: As mentioned in the usecase, we would attach a policy with the object available in Swift storage. The policy would specify who can or cannot access which part of the associated document. The policy would be stored as metadata associated with the object. In the object server, when the object is retrieved, we also retrieve the policy. In our access control module we feed both the requested object and the policy. The response from the access control module is the partial content of the object the requester is allowed to access. Fig. 7 shows the design diagram for our case.

3) Proposed Changes: The proposed changes are only at the object server middleware.

- *Policy file*: We would specify syntax for the policy that Alice can use. The policy would be stored as metadata associated with the object.
- *A/C Module*: The Access Control module takes a input file in JSON format and policy for it along with the credential of the requested user. The result from the A/C module is the partial content of the object the requester is allowed to see.

B. ZeroVM enabled Content Based Access Control for Swift Storage

1) Usecase: ZeroVM enables a user to run arbitrary application on Swift data located inside Swift storage by virtue of a ZeroVM Application. Bob wants to run a ZeroVM application with his own executable (A python script for example) on Alices protected data object ('data.json'). But this time Alice does not worry because she knows a ZeroVM developer is working on Swift Zerocloud midleware that would restrict any arbitrary access on her data object by ZeroVM application owned by other users.

With our proposed design, we would restrict any zerovm application to respect policy set by the owner of a swift data object.

This use case differs from the former case in the sense that in the former case, a user is accessing a Swift object, and in the later case, an application is accessing the object for purpose of arbitrary processing. While in the former case, data object leaves Swift storage, in the later case data object does not actually leave Swift storage.

2) Design Details: In the existing approach, the manifest file of a ZeroVM application contain list of input files, output files and image files possibly containing user application. On receiving exection request, a ZeroVM is launched inside the object server and the user application can arbitrarily access the full content of these files.

In our proposed architecture, we let not allow full content of the Swift objects listed as input to be visible to the user application. In order to achieve this, we introduce the concept of Query. In this case, when the ZeroVM manifest contains a Swift object as input, it also contains predefined queries on that object. In the final manifest for the ZeroVM, the input object is not attached as a channel, instead the temporary files containing the output of the queries are attached as input channels. This is how, we would hide the visibility of the full content of the Swift objects as input. In processing the queries against a input Swift object, we apply the policy associated with the object.

To make our implementation simple, we would use a JSONPath as the a query. While Fig. 8 shows the abstracted view of existing ZeroVM application, our proposed design detail is shown in Fig. 9

3) Proposed Changes: The proposed changes are made at the manifest of the ZeroVM application along with at the object query middleware of the zerocloud.

- *Add query section in the manifest of ZeroVM application*:
- *Query Engine*: The query engine takes a query which is JSONPath in our case, and execute the query against the Swift object. The output of these queries are being saved in temporary files and these file descriptors are feed into the launching ZeroVM as input channels.
- *A/C Module*: We develop an access control module for JSON document which takes a JSON document, a policy file stating which part of the document is accessible by which user, and a set of queries. This module with the help of Query Engine module generates output for these queries.

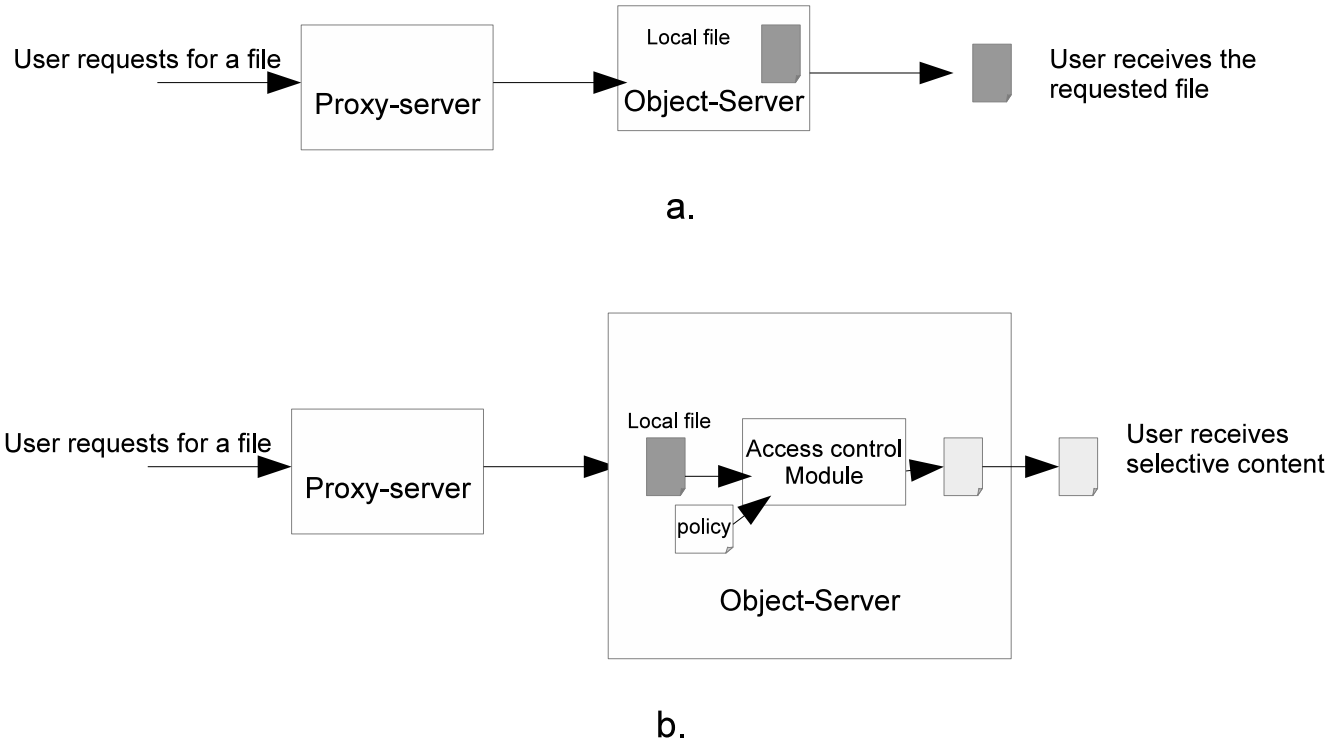

Fig. 7. a. Evaluation of the GET Request for Swift, b. Evaluation of the GET Request for Swift with Proposed Content Filter.

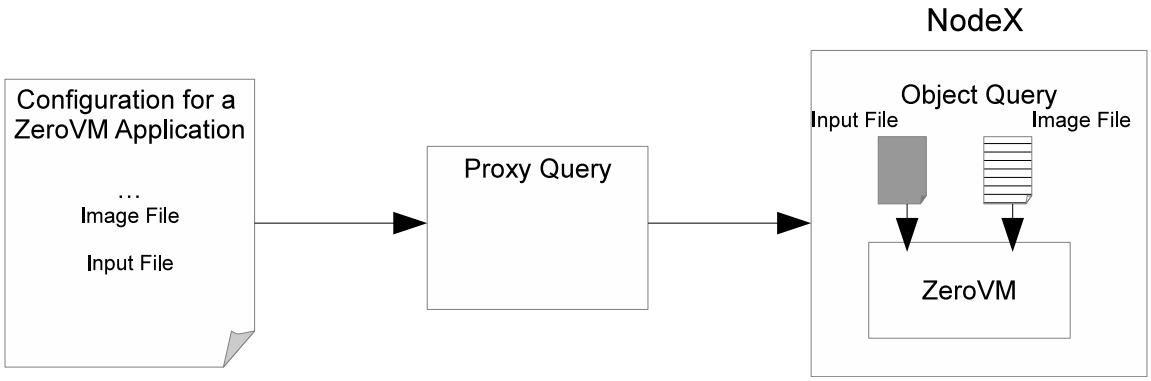

Fig. 8. Abstracted Design for Existing ZeroVM Application .

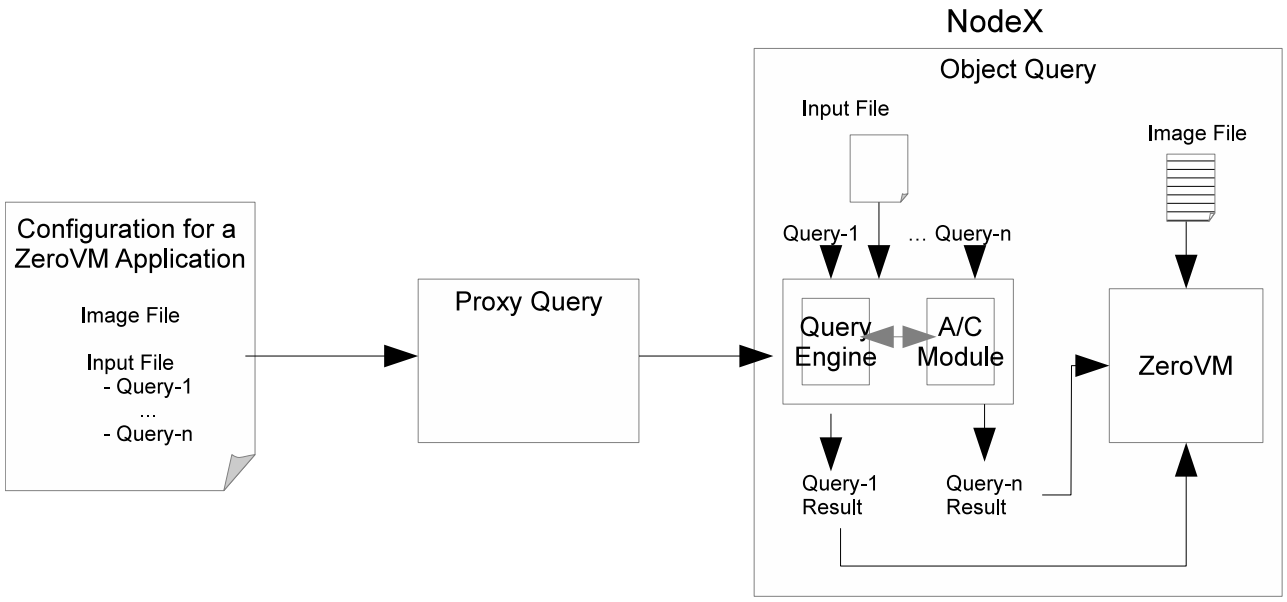

Fig. 9. Design for ZeroVM Application With Proposed Content Filter.