PERMON: An OpenStack Middleware for Runtime Security Policy Enforcement in Clouds

Azadeh Tabiban*, Suryadipta Majumdar*, Lingyu Wang* and Mourad Debbabi*
*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada
Email: {a_tabiba,su_majum,wang,debbabi}@ciise.concordia.ca

***Abstract*—To ensure the accountability of a cloud environment, security policies may be provided as a set of properties to be enforced by cloud providers. However, due to the sheer size of clouds, it can be challenging to provide timely responses to all the requests coming from cloud users at runtime. In this paper, we design and implement a middleware, *PERMON*, as a pluggable interface to OpenStack for intercepting and verifying the legitimacy of user requests at runtime, while leveraging our previous work on proactive security verification to improve the efficiency. We describe detailed implementation of the middleware and demonstrate its usefulness through a use case.**

***Keywords*—Access Control, Event Interception, Middleware, Cloud Security, OpenStack.**

I. Introduction

The multi-tenancy nature of clouds has proved to be a double-edged sword, since it leads to the main advantage of resource optimization while at the same time causes many inherent security concerns [19]. On the other hand, the self-service nature coupled with the sheer size of the cloud renders it a challenging task to enforce tenants' security policies at runtime. In fact, verifying every user event at runtime can cause considerable delays (e.g., over four minutes [3]) even in a mid-sized cloud.

To that end, a promising solution for reducing the response time of security policy enforcement in clouds to a practical level is the proactive approach (e.g., [3], [10], [11]). Such an approach prepares for expensive verification tasks in advance, based on probable temporal relationships between user requests (e.g., a user is likely to create security groups after creating a new VM in OpenStack [15]). Our previous works [10], [11]) have demonstrated that the proactive verification approach may process user requests with only negligible delays.

In this paper, we apply the proactive approach to OpenStack [15], which is one of the most popular cloud platforms, through designing and implementing a middleware, namely *PERMON*. The middleware is designed to first intercept user-issued requests on their path to an intended service. It then identifies the requested event types based on either a single or an aggregation of differentiating fields of corresponding requests. Finally, the middleware processes selected parameters coupled with the identified event types, and enforces the verification results by either granting or rejecting the user requests. Specifically, our main contributions are as follows.

- We apply our proactive verification approach to OpenStack as an efficient middleware solution for enforcing security policies at runtime in clouds.
- We detail the implementation of our approach as a Web Server Gateway Interface (WSGI) pluggable middleware for OpenStack.
- We demonstrate the usefulness of our middleware through describing a use case.
- We address the key challenge of event type identification and discuss the additional benefit of our solution to log processing.

The remainder of the paper is organized as follows. Section II provides background and related work. Section III gives our middleware design. Section IV provides the implementation details. Section V describes a use case. Section VI includes further discussions. Section VII concludes the paper.

II. Preliminaries

In this section, we first provide a quick review of our proactive verification approach [10], [11]) to facilitate further discussions, and then review related work.

A. Proactive Security Verification

Our middleware leverages our previous proactive verification approach, LeaPS [10], [11]. The main idea of LeaPS is to prepare for expensive verification tasks in advance, before actual events are received. Such preparation is possible due to probable temporal relationships between user events, e.g., creating a VM and creating security groups for the same VM, which can be automatically extracted from event logs using machine learning techniques [11]. The results of such preparation (i.e., verification results for possible next events) can then be stored in a table, such that the actual verification performed after events are received amounts to a simple lookup inside the table, which takes far less time than the actual verification does.

To illustrate the idea, Figure 1 compares how user requests are processed under a traditional intercept-and-check approach and under our proactive solution, respectively. In the upper timeline, an intercept-and-check approach intercepts and then verifies the *update port* user request against the desired security property "no bypass" for the anti-spoofing mechanisms in the cloud, which can be violated by real world vulnerabilities, e.g., OSSA-2015-018 [13]. A traditional intercept-and-check approach (e.g., [3]) would take up to several minutes in medium

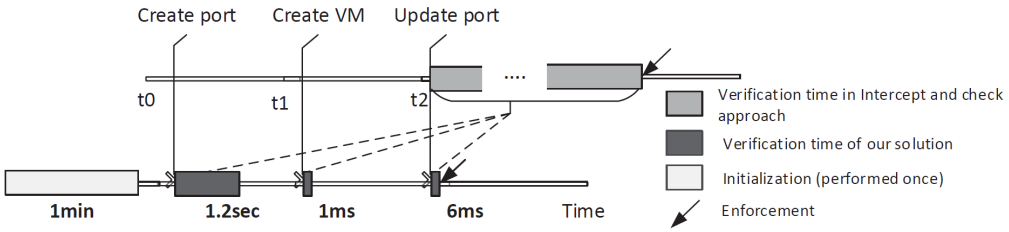

Fig. 1. Different security verification approaches: intercept-and-check (upper) vs. proactive (lower) [10]

size clouds to determine whether the request should be granted or denied, which is apparently unacceptable to the user who has issued the request.

In contrast, as depicted in the lower timeline, the proactive LeaPS approach works very differently: It proactively conducts a set of pre-computations distributed among *N*-steps ahead of the actual occurrence of the critical operation (*Update port*). The pre-computation is based on a dependency model which captures the probable temporal order between user events. The dependency model may be manually created based on structural dependencies inherent to the cloud platform [10], e.g., a security group can only be created after the VM is created. The model can also be automatically established through applying machine learning techniques to extract frequent patterns or sequences of events from the logs, which may correspond to not only aforementioned cloud platform-specific structural dependencies but also other runtime dependencies, e.g., those due to business rules or user habits [11].

The pre-computations incrementally prepare the needed information for efficiently verifying the critical operation later on. Specifically, the verification results of potential future events are pre-computed and stored in a so-called watchlist, and consequently the actual verification only takes negligible time (e.g., six milliseconds [10]) after the actual events are received. In this paper, we design and implement the middleware, *PERMON*, to leverage LeaPS as its proactive verification engine. We will detail the challenges faced in designing *PERMON* to work with LeaPS and provide solutions in later sections.

B. Related Work

There exist many solutions for enforcing security compliance in the cloud. For instance, Solonas et al. [20] propose an approach to detect illegal and undesired activities in the cloud based only on collected billing data in order to preserve privacy. In [12], [9], formal audit approaches have been proposed for security compliance checking in the cloud. Unlike our middleware, those approaches can detect violations only after the fact, which may expose the system to unrecoverable damages. VeriFlow [7] and NetPlumber [6] monitor network events and check network properties and policies at runtime to capture bugs before or as soon as they occur. They rely on incremental calculations to achieve the runtime verification. These works focus on operational network properties (e.g. black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward cloud virtualized infrastructures.

In the context of runtime security monitoring in the cloud, various mechanisms and concepts for designing security service level agreement-based cloud monitoring services have been discussed in [17]. CloudSec [5] and CloudMonatt [25] are solutions that have been proposed for VMs security monitoring. In contrast, our work can potentially cover generic security properties beyond the scope of VMs. In addition, rather than intercepting security measurements, we intercept actual events and assess their impact on the cloud system before applying them. In [16], a host-based secure active monitoring mechanism has been proposed. Unlike our work which focuses on high level policies, the main objective of this work is to detect unwanted low level operations initiated by malicious software.

Retroactive auditing approaches (e.g., [9], [12], [22], [23], [21], [4] in the cloud is a traditional manner to verify the compliance of different components of a cloud. In [9], [12], formal auditing approaches are proposed for retroactive security compliance checking in the cloud. The intercept-and-check approach performs major verification tasks while holding the event instances blocked. Weatherman [3] and OpenStack Congress [14] offer security verification of virtual infrastructure using the intercept-and-check approach, and causes significant delay to a user request. Unlike those works, we provide a proactive enforcement approach which enables starting the verification process in advance.

Proactive security analysis in the cloud is comparatively a new domain with fewer works (e.g., [3], [18], [24], [10], [11]). Weatherman [3] verifies security policies on a future change plan in a virtualized infrastructure using the graph-based model proposed in [2], [1]. Our previous work, PVSC [10], proactively verifies security compliance by utilizing the static patterns in dependency models. Both in Weatherman and PVSC, models are captured manually by expert knowledge. In contrast, this work is an extension of our previous work LeaPS [11], which adopts a learning-based approach to automatically derive the dependency model. Closest to our work, the OpenStack Security Modules (OSM) [8] is an access control framework for OpenStack to support different access control modules by replacing the existing permission checks in OpenStack; the framework includes a service called patron which intercepts user requests similarly as our middleware, whereas the policy enforcement is not proactive like ours.

III. The Design

In this section, we provide the high level design of *PERMON*, a proactive security policy enforcement middleware for OpenStack. The main objective of the middleware is to intercept and verify each user request against given security policies and properties such that the request is either denied or allowed to reach the intended service. We will leverage our proactive verification component LeaPS (see Section II-A), although *PERMON* is designed to work with any other solution which can perform security verification. Figure 2 provides a high level overview about how *PERMON* works. The process consists of four main steps: Users' requests are passed through *PERMON* on their paths to the intended services *(step 1)*. *PERMON* intercepts the requests and extracts parameters which are pre-determined according to the mechanism of the deployed security solution (the proactive verification component LeaPS in our case) *(step 2)*. These parameters are verified by the security solution (LeaPS) *(step 3)*. The verification result on the legitimacy of the requested action is put into effect by *PERMON (step 4)*.

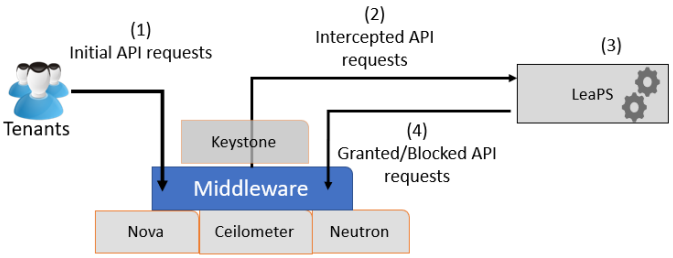

Fig. 2. An overview of *PERMON*

The proactive aspect of *PERMON* works as follows. The main objective is to pre-compute the conditions to be verified at the moment when the so-called critical events (i.e., the events that may directly cause breaches to given security properties) are received. The critical events and those events that are likely to precede the critical events have been pre-defined in the dependency model in advance. At runtime, once *PERMON* intercepts any of those latter events, the verification module will be triggered to precompute the required conditions determined by the imposed security property. When the actual critical event is intercepted by *PERMON*, those precomputed conditions are verified through simple table lookups by analyzing the intercepted attributes, and the legitimacy of the critical event is decided and enforced by *PERMON*.

The main challenges of implementing this design of *PERMON* includes intercepting user requests, identifying types of events, interacting with the verification module, and enforcing the received verification results, which will be detailed in next section.

IV. Implementation

In this section, we detail *PERMON* implementation and its integration to OpenStack. Figure 3 shows the architecture of *PERMON* with examples of inputs to various steps performed by *PERMON*. Algorithm IV describes those steps in more details. In the following, we elaborate on the implementation and our approach of employing the general mechanism of a Web Server Gateway Interface (WSGI) for implementing the middleware functionality.

Algorithm 1 *PERMON* (*APIcalls*, *mark-rulls*, *field-check-rules*)

```
1:  procedure PolicyEnforcement(APICalls)
2:      for each call_i ∈ APIcalls do
3:          env[] = readingEnviron(call_i)
4:          eventToWatch = markFinder(env[], mark-rules)
5:          if eventToWatch is NULL then
6:              response = callNextComponent(call_i)
7:          else
8:              fields = readFields(env, field-check-rules)
9:              Decision = LeaPS(fields, eventToWatch)
10:             if Decision is Allow then
11:                 response = callNextComponent(call_i)
12:             else
13:                 response = callFailed("403 Forbidden")
14:         return response
```

A. Implementing PERMON as a WSGI Middleware

Many features of OpenStack have been implemented as pluggable middlewares based on the Web Server Gateway Interface (WSGI) standard. The main advantage to this approach is that the pluggable nature of deployed middlewares provides developers with flexibility to extend existing functionality without undertaking the cost of customizing existing codes. Following this approach, *PERMON* is designed as a WSGI interface, and is injected into Nova (OpenStack compute service) pipeline alongside other middlewares being stacked up.

Upon receiving clients' requests, each middleware in the pipeline takes the request sent from the previous component, and calls its next adjacent middleware, which means each middleware acts both as a server and an application. Any element that can play the role of a WSGI application has a call method that is passed with two positional arguments when it is executed by the preceding server; the first is a python dictionary for environment information, called *environ*, and the second is a callback function named *start_response*, as detailed below.

- *environ* is a python dictionary that contains contextual information of a request. Such information includes: REQUEST_METHOD, PATH_INFO, wsgi.input, etc. Examples of the REQUEST_METHOD include PUT, GET, POST, etc. PATH_INFO is the URL path an API request is sent to, and wsgi.input is a file-like object and contains data that is sent to the server.
- *start_response* is a callable itself, and takes two positional arguments, status and header; status is an integer followed by a string, and header is a list of tuples that are sent back to the client sending the request.

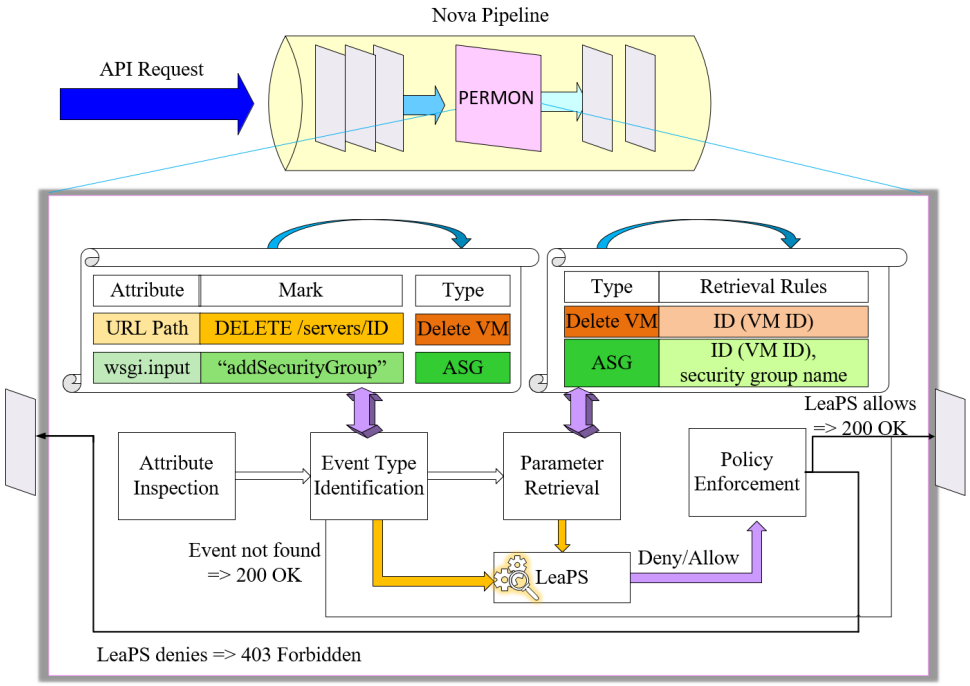

Fig. 3. The Architecture of *PERMON* (the conjunction of METHOD and PATH_INFO is shown as URL PATH; ASG refers to the event type corresponding to adding a VM to a security group)

Having been inserted into the pipeline, *PERMON* is invoked by its preceding element for each request made to Nova, which is passed with the *environ* and *start_response* arguments corresponding to the request. *PERMON* here serves the role of a server for its next element in the pipeline. Inspecting *environ*, we can find useful information to process the corresponding request as we will explain in more details in next section.

B. Event Type Identification

A key challenge in implementing *PERMON* is to identify the intended actions in users' requests intercepted by *PERMON*. The event type corresponding to such actions will determine which actions to be taken while interacting with the proactive verification module. By analyzing OpenStack API documentation, some API requests can be uniquely identified through the conjunction of their METHOD and PATH_INFO attributes. For instance, a given request can be recognized as to delete an instance if and only if its corresponding METHOD and PATH_INFO is DELETE and /servers/*server_id*, respectively.

TABLE I
EXAMPLES OF EVENT TYPES UNIQUELY IDENTIFIED WITH WSGI.INPUT

wsgi.input Content	METHOD	PATH_INFO	Event Name
"{"os-stop": null}"	POST	/servers/*server_id*/action	Stop VM
"{"os-start": null}"	POST	/servers/*server_id*/action	Start VM

However, not all API requests can be uniquely identified through the conjunction of their METHOD and PATH_INFO attributes. For example, as it is shown in Table I, the issued requests for starting and stopping a given server both have the same METHOD and PATH_INFO environmental variables according to API documentation. Therefore, we need another differentiating attribute to fingerprint this group of requests. By studying API documentation, we find that the METHOD field of all these requests is POST. As the data is sent in wsgi.input file-like environmental variable for requests with PUT and POST as their METHOD, we use its content as the differentiating attribute. For example, the content of the wsgi.input object for the two requests (for starting and stopping a given server) is {"os-start": null} and {"os-stop": null}, respectively (Table I), which is intercepted and parsed by *PERMON* as the differentiating attribute.

By parsing the body of requests together with the combination of METHOD and PATH_INFO attributes, PERMON can map them to the type of event they are issued for. This is performed through knowing the collection of differentiating attributes of a given request and a rule mapping these attributes to a pre-defined type of event. To that end, we have embedded inside *PERMON* a set of mapping rules which associate the name of the differentiating attributes and their content with the corresponding event types. For example, the wsgi.input attribute for attaching a VM to security group SG1 is {"addSecurityGroup": {"name":"SG1"}}. To investigate whether a received request is issued to add a VM to a security group, *PERMON* examines wsgi.input attribute, and verifies its starting characters.

As mentioned earlier, we leverage our proactive verification approach, LeaPS, which triggers pre-computation at the occurrence of certain events that are determined to be more likely to

be followed by a critical event. Therefore, our mapping rules for event type identification are designed to map the request attributes to those two types of events. When the former type of events are intercepted, *PERMON* will invoke LeaPS for pre-computation, and when the latter type of events are intercepted, *PERMON* will invoke LeaPS for performing the verification.

C. Interacting with LeaPS and Enforcing the Decisions

Depending on the even types, *PERMON* will invoke LeaPS either for pre-computation or for verification. During such interaction, in addition to the event types, other parameters of the corresponding event need to be extracted and passed to LeaPS in order to verify the legitimacy of the request or to pre-compute conditions. The exact parameters needed are determined based on the security properties and event types. For example, verifying requests against a security property may require the security group name and the VM ID for the event type *add security group*. Therefore, we have implemented inside *PERMON* the mappings between event types and rules to extract required parameters from the intercepted environmental variables of the corresponding request. More specifically, the extraction rules will determine which attributes to be fed into LeaPS, and the way they should be pre-processed by *PERMON*.

Finally, *PERMON* enforces the verification results of LeaPS over the legitimacy of intercepted requests. If LeaPS allows the event, *PERMON* calls the next component in the pipeline and passes to it the intercepted arguments for further processing. If LeaPS indicates denial of the request, *PERMON* blocks the request by controlling the *start_response* function as follows. One of the positional arguments, status, is fed into the callback function *start_response*. The value of this status argument of a request that has passed successfully through the previous component in the pipeline will start with a special value *2xx*. By changing this argument value to *403 Forbidden*, *PERMON* essentially blocks the request and an error message will be sent back to the user.

V. Use Case

In this section, we demonstrate the usefulness of *PERMON* through an example use case. We assume users' requests are made from OpenStack console. The following illustrates the functionalities of *PERMON*, which is to intercept the requests and their parameters, to interact with the verification module LeaPS, and to allow or block the corresponding requests according to the verification result.

The security property to be enforced in the use case is *no downgrade of security group for a running VM*, which prevents attackers from changing the security group of a VM, e.g., from *no_connection* to *essential* (the latter is supposedly a less restricted security group which will allow more connections to the VM, downgrading its security level). For each security group, the verification module LeaPS is initialized with security groups with higher restriction levels. The following is a sample sequence of operations illustrating how our middleware works to enforce the security property.

Stage 1: The first considered user-initiated request is creating a VM, namely leaps_vm.

Stage 2: User requests for attaching the VM to security group *no_connection*. Security group *no_connection* hypothetically implements the most restricted security policy.

The request passes through Nova pipeline and is sent to OpenStack networking service (Neutron) to be executed. Figure 4 shows the corresponding logged request made to Neutron and logged port update event for attaching security group, which means the request has reached Neutron service.

```
2018-03-21 00:36:19.743 4981 INFO neutron.wsgi [req-b95e06bd-4b2c-4a05
-a968-7eebd71e393f f51c874fbe1d4f0c9ce46f60a1e06454
a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 - - [21/Mar/2018
00:36:19] "GET /v2.0/security-groups.json?
fields=id&name=no_connection&tenant_id=a6627ffa0c4f4a3ebaefe05c0b93f4c6
HTTP/1.1" 200 282 0.075747
2018-03-21 00:36:20.498 4981 INFO neutron.wsgi [req-35a4ccc0-48b6-4871
-b316-f333a624a35c f51c874fbe1d4f0c9ce46f60a1e06454
a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 - - [21/Mar/2018
00:36:20] "PUT /v2.0/ports/dfaea8b0-1c1d-4e0b-8fec-d11a8fc5d9bd.json
HTTP/1.1" 200 1084 0.688296
```

Fig. 4. Neutron service logged actions following receiving the request from Nova

Figure 5 shows logged response to the underlined API request, which is associated with attaching to security group *no_connection*. The status_code 202 shows the request has been accepted and successfully processed.

```
2018-03-21 00:36:20.508 5577 INFO nova.osapi_compute.wsgi.server [req-
4ba2a45e-b4a6-4101-88cb-6977f1700d56 f51c874fbe1d4f0c9ce46f60a1e06454
a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 "POST
/v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/57fc4069-065a-4fcb-
b80e-be53286caa99/action HTTP/1.1" status: 202 len: 273 time: 0.9922130
```

Fig. 5. Logged response with status_code 202, associated with request for attaching to security group *no_connection*

Stage 3: The cloud user starts the VM. *PERMON* intercepts the request while it is passing through the pipeline and inspects its body. The request body is the content of wsgi.input of *environ* argument that is sent to the server. If *PERMON* finds the body is associated with starting a VM through verifying wsgi.input attribute of a passing request, it extracts the ID of the VM from another attribute of that request, PATH_INFO. Next, with the extracted ID, LeaPS queries the Neutron database for the security groups leaps_vm is attached to. According to these currently attached security groups, it populates the watchlist with the allowed security groups for the started VM.

```
2018-03-21 01:31:53.485 5576 INFO symcpe_nova.selfservice.middleware
[req-ae420cec-7fcd-44e7-a514-e59dfa0ac39f
f51c874fbe1d4f0c9ce46f60a1e06454 a6627ffa0c4f4a3ebaefe05c0b93f4c6 - -
-] req_body: {"os-start": null}
```

Fig. 6. Intercepted request body for starting VM

Figure 6 illustrates the content of the intercepted parameter, wsgi.input. The format of PATH_INFO and the content of the request corresponding to different server actions are interpreted according to the API documentation as follows. PATH_INFO of a request for starting a VM is in the format

of */servers/server_id/action*. The format provided in the API documentation is used to identify requested actions and to extract different parameters e.g., server_id. As it is shown in Figure 7, we keep track of the running VM with an index to the security groups it can be legitimately attached to.

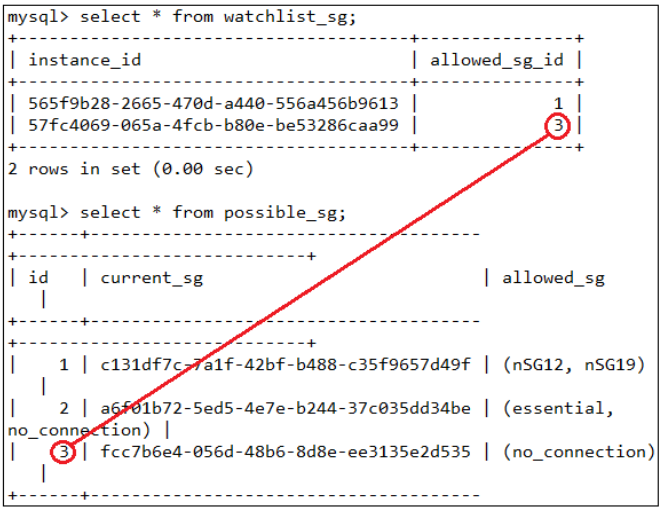

Fig. 7. LeaPS stores IDs of running VMs along with an index to their allowed security groups

Stage 4: At this point, the attacker tries to downgrade the security group of the running VM, leaps_vm, by attaching it to a less restricted security group, namely *essential* (figure 8).

The content of the request is inspected by *PERMON*. Having found the body corresponding to a security group attachment, *PERMON* extracts the security group name and the ID of the VM, which LeaPS looks for and finds among IDs of running VMs in its database. This means that leaps_vm can only be attached to more restricted security groups than what it is currently attached to, i.e., *no_connection*. LeaPS goes through the legitimate security groups indexed by leaps_vm. As security group *essential* is not among them, LeaPS decides on denial of the request.

```
adv@controller:~/OpenStackEnv$ nova add-secgroup leaps_vm essential
ERROR (Forbidden): Forbidden (HTTP 403) (Request-ID: req-85c7368b-
10ae-4a3e-9f70-175f8fc0739f)
```

Fig. 8. Attacker's attempt made to downgrade the security group of the running VM

Consequently, *PERMON* refuses to pass this request to the next element in the pipeline, and prepares its own response with status *403 Forbidden* to be sent back to the client component (Figure 5). Figure 9 shows the logged request body for attaching leaps_vm to security group *essential*. The corresponding logged response is shown in Figure 10 with *status_code* 403, as opposed to 202 in step 2, when the request was accepted and successfully processed. The common request ID between this log entry and the one shown in Figure 9 shows this response corresponds to the request with the content {"addSecurityGroup": {"name":"essential"}}.

```
2018-03-21 02:34:14.502 5576 INFO symcpe_nova.selfservice.middleware
[req-85c7368b-10ae-4a3e-9f70-175f8fc0739f
f51c874fbe1d4f0c9ce46f60a1e06454 a6627ffa0c4f4a3ebaefe05c0b93f4c6 - -
-] req_body: {"addSecurityGroup": {"name": "essential"}}
```

Fig. 9. The logged intercepted body of the API request for attaching to security group *essential*

```
2018-03-21 02:34:14.637 5576 INFO nova.osapi_compute.wsgi.server [req-
85c7368b-10ae-4a3e-9f70-175f8fc0739f f51c874fbe1d4f0c9ce46f60a1e06454
a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 "POST
/v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/57fc4069-065a-4fcb-
b80e-be53286caa99/action HTTP/1.1" status: 403 len: 226 time: 0.1470120
```

Fig. 10. logged response with *status_code* 403, associated with request for attaching to security group *essential*

Furthermore, as opposed to step 2, no log corresponding to attaching security group *essential* can be found in logs of Neutron Service, working synchronized with Nova service, in a time frame close to the same time when the request has been made to Nova, which shows the request has been blocked by our middleware.

VI. Discussions

Benefit to Log Analysis In addition to security policy-enforcement, the *PERMON* middleware can also benefit log processing. In general, OpenStack services only log their responses to the received requests. For log analyses which aim at identifying the triggering requests, those logged responses may not provide sufficient information. Specifically, the log entries can be parsed to extract different fields among which METHOD and PATH_INFO are supposed to map to different types of requested events. However, as mentioned earlier, many events cannot be distinguished from each other. For example, all requests to invoke a server to take some specific action will have the same METHOD and PATH_INFO, which renders log interpretation infeasible. Table II shows three examples of logged responses to such requests. Except for the VM ID at the end of PATI_INFO, which is not useful for identifying the type of requested event, all three log entries have the same combination of METHOD and PATH_INTO. To this end, our *PERMON* middleware logs and examines the intercepted body of requests in order to identify the type of logged requests. This provides a feasible solution for more accurate log processing.

TABLE II
Examples of indistinguishable logged responses corresponding to different requested actions

OpenStack Log Entry	Event Name
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/f6128951-0c48-4a11-8b8b-5e96da77b698/"	Stop VM
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/1223d052-bc35-485a-9237-1830bca80fd7/"	Start VM
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/4c886192-43ad-4f98-90dd-34e24c84fcd0/"	Add Security Group

Discrepancy between Console and GUI Requests By studying intercepted attributes, we note that wsgi.input and METHOD of an issued request can be different depending on whether the request is made from command-line or inside

the OpenStack dashboard. For instance, Table III shows the content of two intercepted attributes logged by *PERMON* when a request for attaching VM1 to security group SG1 is made from command-line as opposed to when it is issued from OpenStack dashboard. The logged attributes are compatible with what is indicated in the API documentation for attaching a VM to a security group in the former case; however, they are mapped to API calls for updating a VM indicated in the API documentation in the latter case. In this work, we set our matching rules according to the API documentation and we expect such discrepancy between internally generated parameters corresponding to identical requests to be addressed in future OpenStack releases.

TABLE III
DISCREPANCY BETWEEN REQUESTS MADE FROM OPENSTACK COMMAND-LINE AND DASHBOARD

	METHOD	wsgi.input	Documented Mapping
Dashaboard	PUT	{"server": {"name": "VM1"}}	Update VM
Command-line	POST	{"addSecurityGroup": {"name": "SG1"}}	Add security group

VII. CONCLUSION

This paper presented a security policy enforcement middleware, *PERMON*, which was designed as a pluggable module in OpenStack Nova service. *PERMON* provided control over user-initiated requests according to given security policies or properties. Working along with our proactive verification module LeaPS, *PERMON* could make decisions about either allowing or denying a request in an efficient manner with only negligible delay to legitimate users. Furthermore, by inspecting the request body to identify requests that are otherwise not distinguishable, *PERMON* could also bring added value to log analysis in OpenStack. Our future work will focus on evaluating the performance of *PERMON* through experiments. Moreover, current implementation of PERMON is OpenStack specific, but we explore the possibility of integrating it to other cloud platforms.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable comments, as well as Yosr Jarraya and Makan Pourzandi for their contributions to the paper. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01566, Discovery Grant N01035 and Prompt Quebec.

REFERENCES

[1] S. Bleikertz, T. Groß, M. Schunter, and K. Eriksson. Automated information flow analysis of virtualized infrastructures. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2011.

[2] S. Bleikertz, C. Vogel, and T. Groß. Cloud Radar: Near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, 2014.

[3] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructure. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, 2015.

[4] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke. Validating cloud infrastructure changes by cloud audits. In *SERVICES'12*.

[5] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. CloudSec: A security monitoring appliance for virtual machines in the iaas cloud model. In *5th International Conference on Network and System Security (NSS)*, 2011.

[6] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[7] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[8] Y. Luo, W. Luo, T. Puyang, Q. Shen, A. Ruan, and Z. Wu. Openstack security modules: A least-invasive access control framework for the cloud. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 51–58. IEEE, 2016.

[9] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[10] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Proactive verification of security compliance for clouds through pre-computation: Application to openstack. In *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[11] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Leaps: Learning-based proactive security auditing for clouds. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 265–285, Cham, 2017. Springer International Publishing.

[12] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Security compliance auditing of identity and access management in the cloud: Application to OpenStack. In *IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[13] OpenStack. Neutron firewall rules bypass through port update, 2015. Available at: https://security.openstack.org/ossa/OSSA-2015-018.html.

[14] OpenStack. OpenStack Congress, 2015. Available at: https://wiki.openstack.org/wiki/Congress.

[15] OpenStack. OpenStack open source cloud computing software, 2015. Available at: http://www.openstack.org.

[16] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy (SP'08)*, 2008.

[17] D. Petcu and C. Craciun. Towards a security SLA-based cloud monitoring service. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*, 2014.

[18] M. Qiu, K. Gai, B. Thuraisingham, L. Tao, and H. Zhao. Proactive user-centric secure data scheme using attribute-based semantic access controls for mobile clouds in financial industry. *Future Generation Computer Systems*, 2016.

[19] K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, (1):69–73, 2012.

[20] M. Solanas, J. Hernandez-Castro, and D. Dutta. Detecting fraudulent activity in a cloud using privacy-friendly data aggregates. Technical report, arXiv preprint, 2014.

[21] K. Ullah, A. Ahmed, and J. Ylitalo. Towards building an automated security compliance tool for the cloud. In *TrustCom'13*.

[22] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE transactions on computers*, 2013.

[23] Y. Wang, Q. Wu, B. Qin, W. Shi, R. H. Deng, and J. Hu. Identity-based data outsourcing with comprehensive auditing in clouds. *IEEE TIFS*, 2017.

[24] S. S. Yau, A. B. Buduru, and V. Nagaraja. Protecting critical cloud infrastructures with predictive capability. In *IEEE 8th International Conference on Cloud Computing (CLOUD)*, 2015.

[25] T. Zhang and R. B. Lee. CloudMonatt: an architecture for security health monitoring and attestation of virtual machines in cloud computing. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.