As strong as the weakest link: Handling compromised components in OpenStack

Aryan TaheriMonfared
Department of Telematics
Norwegian University of Science and Technology
taherimo@stud.ntnu.no

Martin Gilje Jaatun
SINTEF ICT
Trondheim, Norway
Martin.G.Jaatun@sintef.no

***Abstract*—This paper presents an approach to handle compromised components in an Infrastructure-as-a-Service Cloud Computing platform. Our experiments show that traditional incident handling procedures are applicable for cloud computing, but need some modification to function optimally.**

I. Introduction

One of the main obstacles in the movement toward Cloud Computing is its security challenges. Although it has been argued [1] that most of the security issues in Cloud Computing are not fundamentally novel, a new computing model invariably brings its own security doubts and issues to the market.

In a distributed environment with several stakeholders, there will always be numerous ways of attacking and compromising a component, and it is not possible to stop all attacks or to ensure that the system is secure against all threats. Thus, the best approach is to understand impacts and assess the risk of a compromised component. So, we don't study attack methods, instead impacts of a compromised component on the provided service and other components will be analyzed. In order to study impacts of a successful attack, exact functionalities of each component are extracted.

After identifying impacts of a successful attack, we should find efficient approaches to tolerate such an attack and its damages. In this process, the incident should be detected and analyzed first. Detecting and analyzing an incident have a standard procedure that requires knowledge about the normal behavior and operation of the system. The next step is containing the incident.

There are currently several public cloud providers, however none of them disclose their security mechanisms. Thus, we should study applicable mechanisms and introduce new ones to fulfill security requirements of our experimental cloud environment. Publishing these approaches, other researchers can also analyze them and make them more robust.

When we talk about a compromised component in this document, we mean those components in a cloud environment that are disclosed (i.e., private contents revealed), modified, destroyed or even lost. Finding compromised components and identifying their impacts on a cloud environment is crucial.

We have found the OpenStack cloud platform as the best choice for a real case study in our research. In our laboratory configuration, we used the simple flat structure. This will avoid further complexity which is caused by the hierarchical or peer to peer architecture. We have four physical machines, one of them will be the cloud controller, and other three are compute worker nodes. The abstract diagram of our lab setup is depicted in Figure 1.

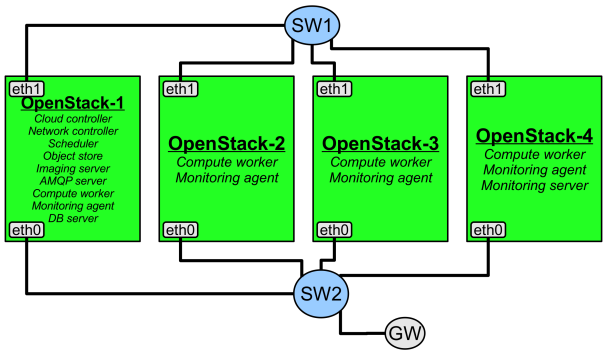

Fig. 1. Lab setup

It should be noted that although we focus on the OpenStack as a specific cloud software in our study, more or less same components and processes may be identified in other cloud platform implementations.

II. Incident handling

We will in the following focus on cloud platform components, functionalities, connected components, access methods and their impacts in case of being compromised. The symptoms of a compromised component are useful in detecting security breaches and must be considered when performing further analysis.

A. Actors' Requirements

Studying the detection and analysis phase of the NIST incident handling guideline [2], and applying new characteristics of Cloud Computing model, we identified several requirements for a cloud provider and a cloud consumer.

1) Cloud providers' requirements:

- **Security APIs:** The cloud provider should develop set of APIs that deliver event monitoring functionalities and also provide forensic services for authorities. Event monitoring APIs ease systematic incident detection for cloud consumers and even third parties. Forensic services

at virtualization level can be implemented by means of virtual machine introspection libraries. An example of an introspection library is XenAccess that allows a privileged domain to access live states of other virtual machines. A cross-layer security approach seems to be the best approach in a distributed environment [3].

- **Precursor or Indication Sources:** The cloud provider deploys, maintains and administrates the cloud infrastructure. The provider also develops required security sensors, logging and monitoring mechanisms to gather enough data for incident detection and analysis at the infrastructure level. As an example, security agents, intrusion monitoring sensors, application log files, report repository, firewall statistics and logs are all part of security relevant indication sources. In case of a security incident, the cloud provider should provide raw data from these sources to affected customers and stakeholders. Thus they will be capable of analyzing raw data and characterizing incident properties.
- **External reports:** The cloud provider should provide a framework to capture external incident reports. These incidents can be reported by cloud consumers, end users or even third parties. This is not a new approach in handling an incident, however finding the responsible stakeholders for that specific incident and ensuring correctness of the incident[1] require extensive research. An illustration, Amazon has developed "Vulnerability Reporting Process"[4] which delivers same functionalities as described before.
- **Stakeholder interaction:** A timely response to an incident requires heavy interaction of stakeholders. In order to ease this interaction at the time of crisis, responsibilities of each stakeholder should be described in detail.
- **Security services:** Cloud consumers may not be interested in developing security mechanisms. The cloud provider can deliver a security service to overcome this issue. Security services which are delivered by the provider can be more reliable in case of an incident and less challenging in the deployment and the incident detection/analysis.
- **Infrastructure information:** When the cloud consumer or another third party wants to develop an incident detection and analysis mechanisms, they may need to understand the underlying infrastructure and its architecture. However, without cloud provider cooperation that won't be feasible. So, the cloud provider should disclose enough information to responsible players to detect the incident in a timely fashion and study it to propose the containment strategy.

2) Cloud consumers' requirements: A cloud consumer must fulfill requirements to ensure effectiveness of the incident detection and analysis process.

- **Consumer's security mechanisms:** The cloud consumer might prefer to develop its own security mechanisms (e.g. incident detection and analysis mechanisms). The customer's security mechanisms can be based on either the cloud provider's APIs or reports from a variety of sources, including: provider's incident reports, end-users' vulnerability reports, third parties' reports.
- **Provider's agents in customer's resources:** By implementing provider's agents, the cloud consumer will facilitate approaching a cross-layer security solution. In this method, the cloud consumer will know the exact amount and type of information that has been disclosed. Moreover, neither the cloud consumer nor the provider needs to know about each others' architecture or infrastructure design.
- **Standard communication protocol:** In order to have a systematic incident detection and analysis mechanisms, it is required to agree on a standard communication protocol that will be used by all stakeholders. This protocol should be independent of a specific provider/customer.
- **Report to other stakeholders:** If the customer cannot implement the provider's agent in its own instances, another approach to informing stakeholders about an incident is by means of traditional reporting mechanisms. These reports should not be limited to an incident only, customers may also use this mechanism to announce a suspicious behavior for more analysis.
- **Cloud consumer's responsibilities:** Roles and responsibilities of a cloud consumer in case of an incident should be defined previously, facilitating immediate reaction in a crisis.

B. Containment of the compromised component

Cloud consumers' allocated resources are not under their direct/physical control. Consumers control their resources using several access methods which may get compromised as well. Specifically in the IaaS service model, the issue is more challenging for responsible organizations (i.e. providers). One of the main reasons is the increased control of a cloud consumer over its allocated resources and virtual instances [5]. The cloud consumer may develop some procedures for containing its service in case of an incident, but applying these procedures is challenging as well. The cloud provider has to ensure that recent changes in the normal operation of a specific service is due to an incident and not a false positive.

We have identified several aspects that should be considered in this phase:

1) We should address the greatest risks and strive for sufficient risk mitigation at the lowest cost, with minimal impact on other mission capabilities [6].
2) The containment, eradication, and recovery should be done in a cost effective fashion. Thus, a cost-benefit analysis of each approach should be performed before application.
3) In a highly distributed system such as a cloud environment, we cannot apply stateful measures, they won't scale.

[1] Avoiding false positive alarms

4) It is not feasible to stop all attacks or secure all components to avoid exploiting any existing vulnerabilities.
5) In addition to the previous item, existing security mechanisms are not completely applicable to the new computing model and they cannot protect the system from all attacks and cannot provide a fast reactive response to an incident.
6) As we cannot harden a cloud environment against all possible attacks, containment strategies and tolerating a successful attack are required approaches.

Our study approach is a case-based one, because:

- Several components, with different functionalities, may require a variety of containment realization mechanisms.
- Providing a single mechanism to cover all incidents, is not possible.
- A combination of mechanisms is possible, and also recommended for covering an attack which exploits several vulnerabilities.
- In each case, we will study different ways of an incident occurrence (e.g. malicious code can be injected in to either a cloud platform service (nova-compute) or OS modules/services.)

C. Case studies

1) Case One: A Compromised Compute Worker: The first case which we will discuss, has only one compromised component. In this case the nova-compute service in the compute worker is compromised, Figure 2.

Two incidents have happened simultaneously in this scenario, malicious code and unauthorized access. The malicious code is injected to the nova-compute service and introduces some misbehavior in it, such as malfunctions in the hosting service of virtual instances, nefarious usage of granted privileges to request for more IP addresses and cause IP address exhaustion in a specific consumer's project.

The malicious code is injected by means of another incident, unauthorized access. The attacker gains access to resources on the OpenStack-4 host, that he/she was not intended to have. Using those escalated privileges, the attacker changed the python code of the nova-compute and restarted the service. Thus, nova-compute started to behave maliciously.

2) Case Two: A bogus component: A bogus service is a threat for the cloud environment security. As the OpenStack is an open source software, an attacker can access the source code or its binaries and deploy a cloud platform service. When the attacker is managing a service, he/she can manipulate the service in a way that threaten the integrity and confidentiality of the environment. This section will discuss such an incident that a bogus cloud platform component is added to the environment. We will focus on a nova-compute service as the bogus cloud platform component.

A bogus nova-compute service or in general any cloud platform component can run on a physical machine or a virtual instance. Adding a physical node to the cloud infrastructure by an attacker, is unlikely; however, for the sake of completeness we study both the case that the bogus service is running on a new physical machine and the one when it is running on a virtual instance. Both cases are depicted in Figures 3, and 4.

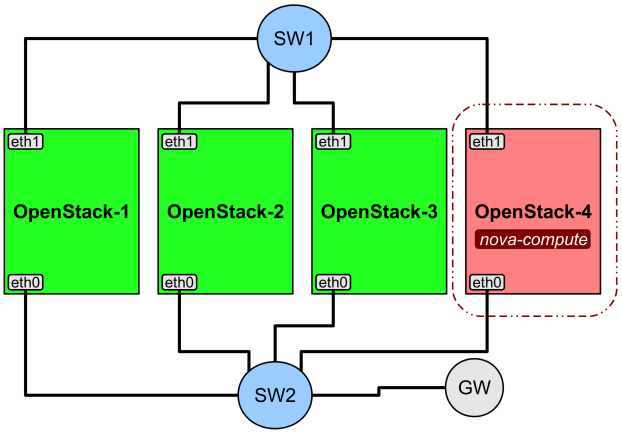

Fig. 2. Case One - The nova-compute service in the OpenStack-4 host is compromised.

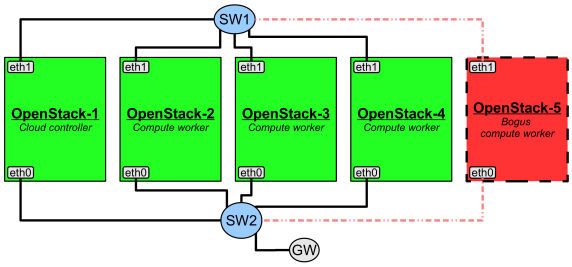

Fig. 3. Case Two - A physical bogus compute worker node is added to the infrastructure.

III. Approaches

We have devised a set of approaches which will be explained in detail in the following.

A. Restricting infected components

A general technique for containing an incident is restricting the infected component. The restriction can be applied in different layers, with a variety of approaches, such as: filtering in the AMQP server, filtering in other components, disabling

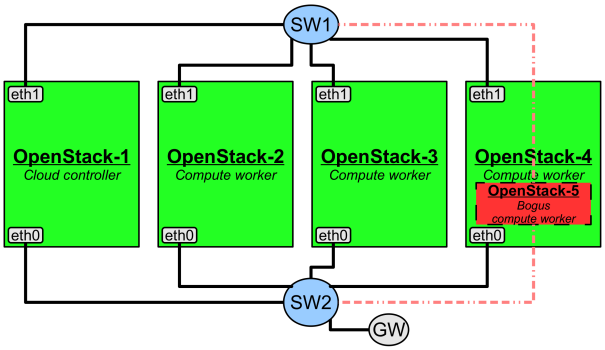

Fig. 4. Case Two - A virtual bogus compute worker is added as a consumer's instance.

the infected service or the communicator one. Additional measures can also be employed to support the restriction, like: removing infected instances from the project VLAN, disabling live migration, or quarantining infected instances.

We explain each of these approaches in the following sections.

1) Filtering in the messaging server (cloud controller): We will propose several filtering mechanisms in the messaging server in order to contain and eradicate an incident in a cloud environment. The OpenStack platform has been used to build our experimental cloud environment. This approach is a responsibility of the cloud provider and the target layer in the cloud platform application layer.

a) Advantages:

- The filtering task at the messaging server level can be done without implementation of new functionalities. We can use existing management interfaces of the RabbitMQ (either CLI or web interface) to filter the compromised component.
- The filtering task can be done in a centralized fashion by means of the management plug-in, although we may have multiple instances of the messaging server.
- Implementing this approach is completely transparent for other stakeholders, such as cloud consumers.
- We can scale out[2] the messaging capability by running multiple instance of the RabbitMQ on different nodes. Scaling out the messaging server will also scale out the filtering mechanism[3].
- This approach is at the application layer, and it is independent of network architecture and employed hardware.
- The implementation at the messaging server level helps in having a fine-grained filtering, based on the message content.

b) Disadvantages:

- A centralized approach has its own disadvantages as well, such as being a single point of failure or becoming the system bottleneck.
- Implementing the filtering mechanism at the messaging server and/or the cloud controller adds an extra complexity to these components.
- When messages are filtered at the application layer in the RabbitMQ server, the network bandwidth is already wasted for the message that has an infected source, destination, or even context. Thus, this approach is less efficient comparing to the one that may filter the message sooner (e.g. at its source host, or in the source cluster)
- Most of the time application layer approaches are not as fast as hardware layer one. In a large scale and distributed environment the operation speed plays a vital role in the system availability and QoS.

  It is possible to use the zFilter technique as a more efficient implementation of the message delivery technique. It can be implemented on either software or hardware. The zFilter is based on the bloom-filter data structure. Each message contains its state; thus this technique is stateless [8]. It also utilizes source routing. zFilter implementations are available for the BSD family operating systems and the NetFPGA boards in the following address, *http://www.psirp.org.*
- Filtering a message without notifying upper layers, may lead to timeout trigger and resend requests from waiting entities. It can also cause more wasted bandwidth.

c) Realization: A variety of filtering mechanisms can be utilized in the messaging server; each of these mechanisms focuses on a specific component/concept in the RabbitMQ messaging server. We can enforce the filtering in messaging server *connection*, *exchange*, and *queue* that will be discussed next.

- **Connection:** A connection is created to connect a client to an AMQP broker [9]. A connection is a long-lasting communication capability and may contain multiple channels [10]. By closing the connection all of its channels will be closed as well.
- **Exchange:** An exchange is a message routing agent which can be durable, temporary, or auto-deleted. Messages are routed to qualified queues by the exchange. A Binding is a link between an exchange and a queue. An exchange type can be one of *direct, topic, headers,* or *fanout*. [11]

  An exchange can be manipulated in different ways in order to provide a filter mechanisms for our cloud environment:
  - **Unbinding a queue from the exchange:** The compromised component queue won't receive messages from the unbinded exchange.
  - **Publishing a warning message:** Publishing an alert message to that exchange, so all clients using that exchange will be informed about the compromised component. Thus, by specifying the compromised component, other clients can avoid communicating with it. The main obstacle in this technique is the requirement for implementing new functionalities in clients.
  - **Deleting the exchange:** Deleting an exchange will stop routing of messages related to it. It may have multiple side effects, such as memory overflow and queue exhaustion.
- **Queue:** Queue is called as a "weak FIFO" buffer, that each message in it can be delivered only to a single client unless re-queuing the message [11].
  - **Unbinding** a queue from an exchange avoids further routing of messages from that exchange to the unbind-ed queue. We can unbind the queue which is connected to the compromised component and stop receiving messages by the infected client.
  - **Deleting** a queue not only removes the queue itself, but also remove all messages in the queue and cancel

[2]Scaling out or horizontal scaling is referred to the application deployment on multiple servers [7].

[3]But it may require a correlation entity to handle the filtering tasks among all messaging servers.

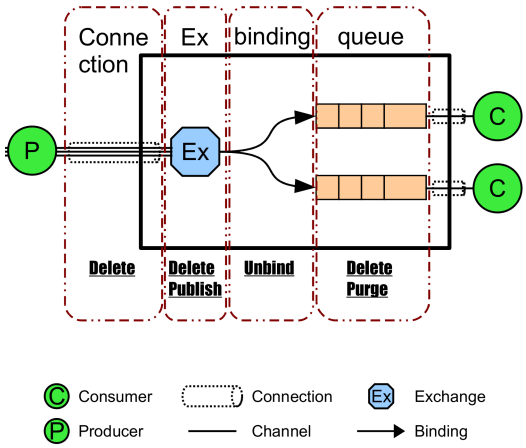

Fig. 5. Overview of RabbitMQ messaging server and applicable containment approaches.

all consumers on that queue.

– **Purging** a queue removes all messages in the queue that do not need acknowledgment. Although it may be useful in some cases, it may not be as effective as required in occurrence of an incident.

Figure 5 depicts a simplified overview of messaging server internal entities and the application points of our approaches.

2) Filtering in each component: Applicable filtering mechanisms in the messaging server have been studied in the previous section. This section discusses mechanisms that are appropriate for other components. These components are not essentially aware of messaging technique details and specifications.

a) Advantages:

- The implementation of the filtering mechanism in each component avoids added complexity to the messaging server and cloud controller.
- This approach is a distributed solution without a single point of failure in contrast to the previous one with a centralized filtering mechanism.
- Assuming locality principle in the cloud, wasted bandwidth is limited into a cluster/rack which host the infected components. Network connections have much higher speed in a rack or cluster.
- This approach does not require a correlation/coordination entity for filtering messages. Each component behaves independently and autonomously upon receiving an alarm message, that announces a compromised node.

  As there is no boundary in the cloud, performing security enforcement at each component is a more reliable approach. Traditionally, most security mechanisms have been employed at the organization/system boundaries. However, as the realization of boundaries is becoming weaker in a cloud environment, this approach is a reasonable one to fulfill the new requirements.

b) Disadvantages:

- When the filtering must be performed in each component, all interacting components must be modified to support the filtering mechanism. However, this issue can be relaxed by using a unified version of messaging client (e.g. pika python client) and modifying the client in case of new requirements.
- The message which should be discarded traverses all the way down to the destination, and wastes the link bandwidth on its route.
- Dropping a message without notifying upper layers, may lead to timeout trigger and resend requests from waiting entities. It can also cause more wasted bandwidth.

c) Realization: This approach can be implemented at two different levels: blocking at either the messaging client level (e.g. AMQP messaging client) or the OpenStack component/service level.

First, the responsible client can be modified to drop messages with specific properties (e.g. infected source/destination). As an example, the responsible client for AMQP messaging in the OpenStack is amqplib/pika; we must implement the mechanism in this AMQP client (or its wrapper in the OpenStack) to filter malicious AMQP messages. Using this method, more interaction between the OpenStack and clients may be required to avoid resend requests. Because of using the same AMQP client in all components, the implementation is easier and its modification process needs less effort.

The second method is to develop the filtering in each of the OpenStack components, such as nova-compute, nova-network, nova-scheduler, etc. This method adds more complexity to those components and it may not be part of their responsibilities.

We propose a combination of these methods. Implementing the filtering mechanism in the carrot/amqplib wrapper of the OpenStack has advantages of both methods and avoids unnecessary complexity. The OpenStack wrapper for managing AMQP messaging is implemented in *src/nova/rpc.py*. In order to identify the malicious message, we use the message address which is part of its context. Then, the actual dropping happens in the *AdapterConsumer* method. Assuming that the source address is set in the context variable, filtering is straight forward. By checking the message address and avoiding the method call, most of the task is done. The only remaining part is to inform the sender about the problem, that can be implemented by means of the existing message reply functionality.

3) Disabling services: Disabling services is a strategy for containing the incident. The disabled service can be either the infected or the communicator one. The communicator service handles tasks distribution and delegation. This method can be used only by the cloud provider, and is at the application layer.

a) Disabling an infected service: An incident can be contained by disabling the infected service. It has several advantages, including:

- After stopping the nova-compute service, running instances will continue to work. Thus, as a result con-

sumers' instances will not be terminated nor disrupted.
- All communications to and from the compromised node will be stopped. So, the wasted bandwidth will reduce massively.
- Shutting down a service gracefully, avoids an extra set of failures. When the service is stopped by Nova interfaces, all other components will be notified and the compromised node will be removed from the list of available compute workers.

Like any other solution, it has multiple drawback as well, including:

- Keeping instances in the running status can threaten cloud consumers. The attacker may gain an access to running instances on the compromised node.
- The live migration feature will not work anymore. Thus, the threatened consumers cannot migrate running instances to a safe or quarantine compute worker node.
- Neither the cloud provider nor consumers can manage running instances through the OpenStack platform.

This approach requires no further implementation, although we may like to add a mechanisms to turn services on and off remotely.

b) Disabling a communicator service: An incident can be contained by disabling or modifying its corresponding communicator service. An example of a communicator service in an OpenStack deployment nova-scheduler service. The nova-scheduler decides that which worker should handle the newly arrived request, such as running an instance.

By adding new features to the scheduler service, the platform can avoid forwarding request to the compromised node.

Advantages of this approach are:

- No more requests will be forwarded to the compromised node.
- Consumers' instances remain in the running status on the compromised node. So, consumers will have enough time to migrate their instances to a quarantine worker node or dispose their critical data. Even estimate impacts of the incident.
- This approach can be used to identify the attackers, hidden system vulnerabilities, and the set of employed exploits. In other words, it can be used for forensic purposes.

And its disadvantages are:

- New features should be implemented. These new features are more focused on the decision algorithm of the scheduler service.
- This approach will not secure the rest of our cloud environment, but it avoids forwarding new requests to the compromised node. However, this drawback can be seen as an opportunity. We can apply this approach and also move the compromised node to a **HoneyCloud**. In the HoneyCloud we don't restrict the compromised node, instead analyze the attack and attacker's behavior. But even by moving the compromised node to a HoneyCloud, hosted instances on that node are still in danger. It is possible that consumers' instances are all interconnected. Thus, those running instances, on the compromised node in the HoneyCloud, threaten the rest of consumers' instances. The rest of instances may even be hosted on a secure worker node. The next proposed approach is a solution for this issue.

4) Removing instances from the project VLAN: This approach does not contain the compromised node, instead focuses on containing instances hosted by the compromised worker node. This is important because those instances may have been compromised as well. The first step toward securing the consumer's service is to disconnect potentially infected instances.

The main usecase of this approach is when the attacker disrupts other solutions (i.e. disabling nova-compute management functionalities, escalated privileges at the OS layer), or when instances and the consumer's service security is very important (e.g. eGovernment services).

It has several advantages specifically for cloud consumers, including:

- Disconnect potentially infected instances from the rest of consumer's instance.
- It does not require new features implementation.
- The attacker cannot disrupt this method.

And its disadvantages are as follows:

- This method only works in a specific OpenStack networking mode (i.e. VLANManager networking mode).
- The consumer completely loses control over isolated instances, that may lead to data loss or disclosure, service unavailability, etc.

5) Disabling live migration: Live migration can cause wide-spread infection, or can be a mechanism for further intrusion to a cloud environment. It may take place intentionally or unintentionally (e.g. an affected consumer may migrate instances to resolve the attack side effects, or the attacker that has the consumer privileges migrates instances to use a hypervisor vulnerability and gain control over more nodes). Disabling this feature helps the cloud provider to contain the incident more easily, and keep the rest of the environment safer.

6) Quarantining instances: When we migrate instances from a compromised node, we cannot accept the risk of spreading infection along instance migration. Thus, we should move them to a quarantine worker node first. The quarantine worker node has specific functionalities and tasks, including:

- This worker node limits instances connectivity with the rest of cloud environment. As an example, only cloud management requests/responses are delivered by the quarantine host.
- It has a set of mechanisms to check instances' integrity and healthiness. These mechanisms can be provided by the underlying hypervisor, cloud platform, or third parties' services.

B. Replicating services

An approach to overcome the implications of an incident is replicating services. A service in this section is a service which is delivered and maintained by the cloud provider. It can be a cloud platform service (e.g nova-compute) or any other services that concerns other stakeholders. The replication can be done passively or actively, and that is due to new characteristics of the cloud model. The replication of a cloud service can be done either at the physical or virtual machine layer.

1) Replicate services on physical machines: Replicating service on physical machines is already done in a platform such as the OpenStack. The provider can replicate cloud services either passively or actively when facing an issue in the environment.

2) Replicate services on virtual machines: Replication of service on virtual machines has multiple benefits, including:

- Virtual machines can be migrated while running (i.e. live migration), this is a practical mechanism for stateful services that use memory.
- Replication at the instance layer is helpful for forensics purposes. It is also possible to move the compromised service in conjunction with the underlying instance to a HoneyCloud. This is done instead of moving the physical node, ceasing all services on it, and changing the network configuration in order to restrict the compromised node communication.
- Using virtual machines in a cloud environment we can also benefit from the cloud model elasticity and on demand access to computing resources.

This approach is also the main idea behind the CC-VIT [12]. By applying the CC-VIT to our environment, the preferred hybrid fault model will be REMH, and the group communication is handle using the AMQP messaging.

We can use physical-to-virtual converters to have the advantages of both approaches. These tools convert a physical machine to a virtual machine image/instance that can be run on top of a hypervisor.

Moreover, each of these replicas can be either active or passive. This will have a great impact on the system availability.

C. Disinfecting infected components

Disinfecting an infected component is a crucial task in handling an incident and securing the system. It can be accomplished with multiple methods having a variety of specifications.

None of the following approaches will be used for cleaning the infected binary files, instead less complex techniques are employed that can be applied in a highly distributed environment. Cleaning a binary file can be offered by a third party security service provider, that has focused on large scale antivirus software.

1) **Updating the code**
   The service code can be updated to the latest, patched version. This process should be done in a smooth way so all components will be either updated or remain compatible with each other after partial components update. Several tools has been developed with this purpose. One of the best examples is the Puppet project [13].
2) **Purging the infected service**
   Assuming that the attacker has stopped at the cloud platform layer, by removing the service completely we can assure containment of the incident.
3) **Replacing the service**
   Another method which is not as strong as others, is achieved by replacing the infected service with another one that uses a different set of application layer resources, such as configuration files, binaries, etc. Thus, we can be sure that the infected resources have no effect on the new service.

D. Migrating instances

The affected consumer can migrate an specific instance or a set of instances to another compute worker or even another cloud environment. The migration among different provider is an open challenge nowadays, because of the weak interoperability of cloud systems and lack of standard interfaces for cloud services.

In our deployment, both Amazon EC2 APIs and RackSpace APIs are supported. Thus, in theory a consumer can move between any cloud environment provided by the Amazon EC2, RackSpace, and any open deployment of OpenStack without any problem.

E. Node authentication

In this method each worker must have a certificate signed by a trusted authority. This authority can be either an external one or the cloud controller/authentication manager itself. Having a signed certificate, the worker can communicate with other components securely. The secure communication can bring us any of the following: confidentiality, integrity, authentication, and non-reputation.

In this case, worker's communication and authenticity is important for us. For this purpose we can use two different schemes: message encryption or a signature scheme. Each of these schemes can be used for the whole communication or the handshake phase only.

When any of those schemes are applied only to the handshake phase, any disconnection or timeout in the communication is a threat to the trust relation. As an authenticated worker is disconnected and reconnected, we cannot only rely on the worker's ID or host-name to presume it as the trusted one. Thus, the handshake phase should be repeated to ensure the authenticity of the worker.

Although applying each scheme to all messages among cloud components is tolerant against disruption and disconnection, its overhead for the system and the demand for it should be studied case by case.

By applying each of those schemes to all messages, we can tolerate disconnection and disruption. However, using cryptographic techniques for all messages introduce an overhead for the system which may not be efficient or acceptable.

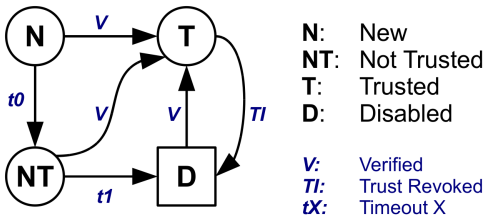

Fig. 6. A sample markov model for trust states of a component.

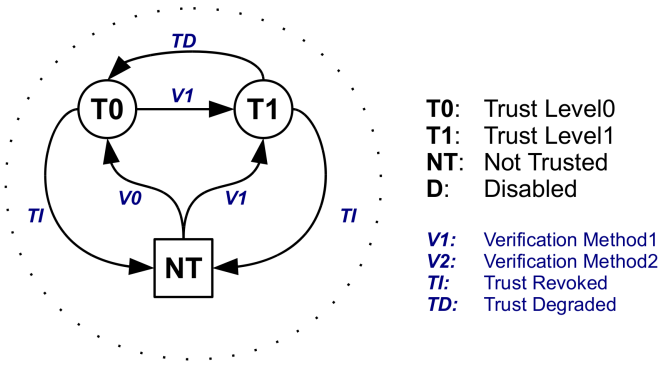

Fig. 7. A sample markov model for transitions between different trust levels of a component.

F. Policies

1) No new worker policy: In addition to all those technical approaches, a set of management policies can also relax the issue. As an example, no new worker should be added unless there is a demand for it. The demand for a new worker can be determined when the resource utilization for each zone is above a given threshold.

2) Trust levels and timeouts: Introducing a set of trust levels, a new worker can be labeled as a not trusted worker. Workers which are not trusted yet, can be used for hosting non-critical instances, or can offer a cheaper service to consumers.

In order to ensure the system trustworthiness in a long run, a not-trusted worker will be disabled after a timeout. A simple Markov model of those transitions are depicted in Figure 6.

Assuming we have only two trust levels, Figure 7 depicts transitions between them. As an example, *T0* can be achieved by human intervention; and the second level of trust *T1* is gained by cryptographic techniques or trusted computing mechanisms.

3) Manual confirmation: In this method, recently added workers are not used for serving consumers' requests until their authenticity is confirmed by the cloud provider. This method requires human intervention; thus, it can become a bottleneck in the cloud infrastructure. Techniques, explained in the next part, can relax the bottleneck issue.

IV. Conclusion

We have presented an approach to handling compromised components in an OpenStack IaaS configuration. Cloud Computing present some unique challenges to incident handling, but our experience shows that with proper adaptation, traditional incident management approaches can also be employed in a Cloud Computing environment.

Acknowledgment

This paper is based on results from MSc Thesis work performed at NTNU.

References

[1] Y. Chen, V. Paxson, and R. H. Katz, "What's New About Cloud Computing Security?" EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-5, Jan 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.html

[2] T. G. Karen Scarfone and K. Masone, "Computer Security Incident Handling Guide," NIST, Special Publications SP 800-61 Rev. 1, March 2008, http://csrc.nist.gov/publications/nistpubs/800-61-rev1/SP800-61rev1.pdf.

[3] A. TaheriMonfared and M. G. Jaatun, "Monitoring Intrusions and Security Breaches in Highly Distributed Cloud Environments," in *Proceedings of CloudCom 2011*, November 2011.

[4] AWS Security Team, "Vulnerability Reporting," http://aws.amazon.com/security/vulnerability-reporting/, March 2011.

[5] J. Reed, "Following Incidents into the Cloud," SANS Institute, Security Reading Room, 2011, http://www.sans.org/reading_room/whitepapers/incident/incidents-cloud_33619.

[6] G. Stoneburner, A. Goguen, and A. Feringa, "Risk Management Guide for Information Technology Systems," National Institute of Standards and Technology, Special Publications, July 2002.

[7] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski, "Scale-up x scale-out: A case study using nutch/lucene," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, march 2007, pp. 1 –8.

[8] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, ser. SIGCOMM '09. New York, NY, USA: ACM, 2009, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592592

[9] "Rabbitmq core api guide," http://www.rabbitmq.com/api-guide.html, May 2011.

[10] C. Trieloff, C. McHale, G. Sim, H. Piskiel, J. O'Hara, J. Brome, K. van der Riet, M. Atwell, M. Lucina, P. Hintjens, R. Greig, S. Joyce, and S. Shrivastava, "Advanced message queuing protocol protocol specification," AMQP.org, amq-spec, July 2006, version 0.8.

[11] D. Samovskiy, "Introduction to AMQP Messaging with RabbitMQ," July 2008.

[12] Y. Tan, D. Luo, and J. Wang, "CC-VIT: Virtualization Intrusion Tolerance Based on Cloud Computing," in *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, December 2010, pp. 1 –6.

[13] "Puppet labs," http://www.puppetlabs.com/, May 2011.