A Framework Architecture for Agentless Cloud Endpoint Security Monitoring

Asem Ghaleb[1], Issa Traore[1], Karim Ganame[2]

***Abstract*— Cloud computing endpoints security monitoring faces more challenges compared with traditional networks due to the ephemeral nature of cloud assets. Existing endpoint security monitors use agents that must be installed on every computing host or endpoint. However, as the number of monitored instances increases, agents installation, configuration and maintenance become arduous and requires more efforts. Moreover, installed agents can increase the security threat footprint and several companies impose restrictions on using agents on every computing system. This work provides a generic agentless endpoint framework for security monitoring of cloud computing endpoints. The endpoints are accessed by the monitoring framework running on a central server. Since the monitoring framework is separate from the machines for which the monitoring is being performed, the various security models of the framework can perform data retrieval and analysis without utilizing agents executing within the endpoints. The monitoring framework retrieves transparently raw data from the monitored endpoints that are then fed to the security modules integrated with the framework. These modules analyze the received data to perform security monitoring of the target endpoints. As a use case, a real-time intrusion detection model has been implemented to detect abnormal behaviors on endpoints based on the data collected using the introduced framework.**

Keywords: *Cloud security monitoring, agentless monitoring, software framework, anomaly detection.*

I. Introduction

In the last few years, significant technological advancement has been made on monitoring, for cloud infrastucture, common quality of services (QoS), such as performance, scalability, and availability. Several tools delivered by cloud hosting companies or third party vendors exist in the market, allowing monitoring and tracking service level agreements (SLAs) for the above-mentioned types of QoS. In contrast, the area of cloud security monitoring is still a technological desert, with only a handful of available tools suited for the specific challenges inherent to cloud computing. In particular, cloud endpoint security monitoring faces several technical challenges due to the dynamic and ephemeral nature of cloud resources. The traditional way of monitoring computing hosts, referred to as agent-based approach, often requires the installation of agents software on the target hosts that periodically scan targeted hosts and collect data about the hosts or the applications running on hosts [2]. The collected data are either transferred to a management server for analysis or analyzed by a management software locally on the same host. However, this approach of monitoring suffers several scalability, security and performance drawbacks, and those drawbacks affect user acceptability. There is a need to install agent software and perform detailed configuration for each host [4], and the installed agents require continuing maintenance. In addition, the use of agents impacts deployment time on computing hosts, as agents should be installed and their updates should be thoroughly checked and applied before deployment on the endpoints. Thus, endpoints may get compromised if updates are not applied regularly and when new hosts are plugged in the network before having updated agents installed on them. With regards to security, the installed agents on hosts may involve vulnerabilities which increase the attack surface of the monitored hosts. Attackers can target the installed agents and their running services and take advantage of the privileges granted to the agents. Hence, compromising the agents enables attackers to get control of the hosts monitored by the agents. Another major drawback of agent-based approach appears when there are several virtual machines running on an endpoint. In this case, there would be a need to install agents on every virtual instance besides the agent installed on the hosting machine itself. This could impact the functionality and the performance of the endpoints, as each agent running on every instance consumes some resources out of the total computing host resources. Furthermore, cloud elasticity and the constant reconfiguration and migration of cloud resources make the deployment and maintenance of agents on coud endpoints costly and cumbersome. An agentless approach, on the other hand, collects data from endpoints without installing any agent on the instances being monitored [2]. This makes agentless approach easy to manage than agent-based approach. However, agentless monitoring raises a number of challenges (e.g., security, scalability, and performance), and to our knowledge limited research has been done in this area to this date.

The objective of our work is to address these challenges by proposing a new agentless framework for security monitoring of cloud computing endpoints. The monitoring process should be achieved in an efficient, low latency, scalable, and secure way by utilizing regular communication protocols that are often pre-installed and standard on most machines. The aim is to use the proposed framework for collecting host-based data and integrate it with anomaly-based intrusion detection models. The rest of the paper is organized as follows. Section II provides a review of the related work

[1]A. Ghaleb is with University of Victoria, ECE Department, Victoria, BC Canada aghaleb@uvic.ca

[1] I. Traore is with University of Victoria, ECE Department, Victoria, BC Canada itraore@ece.uvic.ca

[2]K. Ganame is with StreamScan, 2300 Rue Sherbrooke E, Montreal, QC, Canada ganame@streamscan.io

in the areas of agentless security monitors and detectors. Section III introduces our agentless endpoint security monitoring framework along with its architecture, design and implementation details. Section IV presents the emprical evaluation of the proposed framework. Section V presents a use case involving a real-time intrusion detection model. Section VI summarizes the contributions of the current work, and discusses possible research directions for future work.

II. Related Work

The concept of agentless monitoring has been used for building monitors and malware/intrusion detectors, especially in the cloud and virtual machines. In this section we review related research for agentless monitoring and present the findings in this field.

Cui et al. [5] proposed an agentless architecture for monitoring processes running on virtual machines in a cloud environment. They developed a proof of concept by developing a modified KVM kernel and OpenStack plugins.

Berlin et al. [3] investigated the potential of using an agentless utility for detecting malicious endpoint behavior based on Windows audit logs as a supplement for the existing defense tools. They have setup a data collector to collect events of the file/registry's writes, deletes and executes, and processes spawned. The events are collected by running malicious and benign samples in a sandbox for a time-window of 4 minutes. A Logistic Regression (LR) model has been trained using the features extracted from the collected events. However, the proposed approach deals only with events related to file/registry's writes, deletes and executes, and processes spawned. The proposed models can not be expanded by users to integrate other audit logs.

Tang et al. [8] proposed an agentless antivirus system (VirtAV) for Antivirus protection on VMs based on in-memory signature scanning. To prevent attacks in the guest VM from reaching the antivirus system, in this work, the monitoring of events and detection of virus are offloaded to the hyeprvisor or virtual machine monitor (VMM). VirtAV-engine searches for viruses' signatures by scanning the host memory for footprints of executable in guest VMs. A prototype has been implemented based on Qemu/KVM hypervisor. The evaluation experiments based on 3546 samples of viruses showed that the approach can find all the sample viruses, and acceptable overhead is introduced to the VM.

Brattstrom et. al [4] proposed scalable and agentless network monitoring system. The monitoring system combines a collector, a time series database and a dashboard running on 'plug-and-play'Pi. In their implementation, each Raspberry Pi is loaded with a Docker image of the system. The Pi uses Simple Network Management Protocol (SNMP) to poll the monitored network devices. The collected data is stored in the database and presented on Grafana dashboard. The authors claim that adding more Pi devices to the network supports the scalability of the system. The paper did not discuss the types of data that can be collected. The system just presents data analytics on the dashboard. Unlike our work, the data in the time series database is not to be accessed and used by other security defense models. The proposed system is not generic, as the extension of the polled data requires modification of the collector deployed on the Pi devices as part of the preconfigured Docker image.

III. Framework Architecture

In this section, we present the implementation details of the proposed framework. First, the general architecture of the framework is introduced. Then it proceeds with discussing the development details subsequently in the following subsections.

A. Framework Design

Before starting with the design details of the proposed framework, we need to specify the requirements of this framework. Following is a summary list of the framework main requirements:

1) The need for a security monitoring framework that can be used for monitoring endpoint devices while eliminating the need for installing monitoring agents on the instances being monitored.
2) The framework should provide a level of security to protect against different kinds of possible security attacks.
3) The framework should be scalable in a way that enables the monitoring of large number of endpoints while maintaining the scanning performance.
4) The framework should perform the data retrieval with low network latency so that it can be used to scan large number of instances with no impact on the network performance.
5) The framework should be designed to involve flexible, and easy-to-use components.

The design phase will state the architecture of the framework and the main components constituting the framework based on the specified requirements. Finally, the development phase of the framework will be carried out.

Figure 1 depicts the framework main components and how they communicate with each other. A brief description of each component is provided as follows.

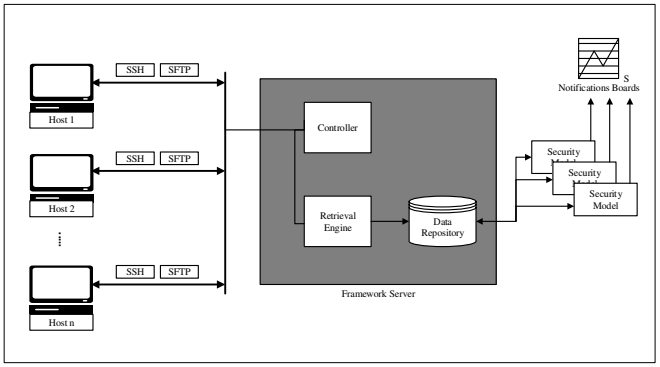

Fig. 1. Framework architecture

- **Controller**
  The controller is a management service that manages and establishes connections with the monitored endpoints on a regular basis or on request. In addition, it is necessary to make sure that the connections and the data gathering are performed in a secure way to protect against attacks targeting data confidentiality, integrity, and availability.
- **Retrieval engine**
  The Retrieval engine is the main component of the framework in charge of collecting host-based data from the monitored hosts. The framework is supposed to provide a configuration means, embedded within this engine, enabling the security officers or the framework administrators to specify the list of endpoints to be monitored, in addition to determining the different host-based data items that should be retrieved from each monitored instance. This engine is not supposed to establish direct connections with the monitored endpoints. Scanning requests should be forwarded to the controller that will establish the connections accordingly in a secure way.
- **Data repository**
  The collected data from the monitored endpoints are stored in a storage media (e.g. log files, light database, etc.) in a special format. This storage will work as a shared repository that can be accessed by various security scanning modules, integrated with the framework, at the same time in one-to-many relationship, which may enhance the performance and the accuracy of the monitoring process.
- **Security models**
  The main purpose of the proposed framework is the monitoring of the target endpoints. As mentioned in the previous sections, the framework components will work on collecting various host-based data items that will be used as a feed to the security models which are responsible for distinguishing normal behavior from suspicious and malicious ones. In this work, we are not going to propose any security models and the framework can be integrated with existing models targeting various types of malicious activities.
- **Notification board**
  The results of the security checks done by the security scanners integrated with the framework can be displayed to the operators or security officers via the notification boards, particularly the critical alarms and a summary of the last scanning routines.
- **Employed protocols**
  For the sake of interoperability, the framework is implemented in such way that it can be used to monitor different instances running on different platforms (e.g., Linux, Windows, etc.). The framework connects with the monitored endpoints and collects data items by using standard protocols, such as SSH, SFTP, and so on, that are supported by most operating systems.

The suggested work flow of the proposed framework is shown in Figure 2. The first step "**Identify the host IP to scan**" decides which instance or group of instances will be scanned at the current timestamp. This can be modeled using different selection criteria including priority, last scan time, etc. The second step "**Determine the data items to collect**" works on selecting the different host-based data items that will be collected for the endpoints identified in Step 1 and the related scripts/code that will be executed on the target endpoints to collect the specified data items. Steps 1 and 2 are performed by the Retrieval engine. In step 3 "**Establish secure connection,**" the controller will receive a request with the target endpoint information. Based on this request, the controller will establish a secure connection with the specified endpoint and informs the retrieval engine upon success which will accordingly send subsequent requests representing the data items to be collected. The controller will execute the corresponding commands/scripts to collect the required host-based data items as in step 4 "**Collect data items**." In the last step 5 "**Store retrieved data**," the collected data will be stored in the data repository. This process is repeatedly executed on regular basis to keep the endpoints monitored up to the moment.

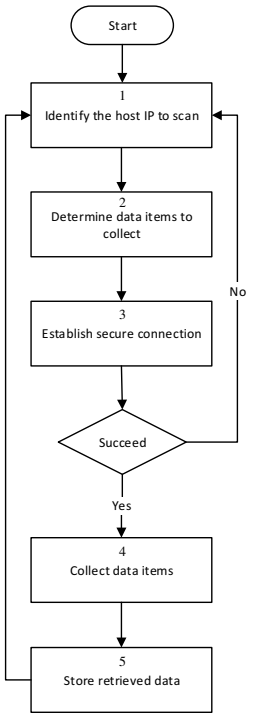

Fig. 2. Work flow

B. Framework Implementation

According to the framework architecture depicted in Figure 1, the framework consists of a set of core components which handle raw data collection and processing, and another set of components that manage the collection process. This section will present the technical details of all framework components and how they are implemented.

1) Collection process management: The framework provides three configuration files, which enable the security officers or the framework administrators to maintain various framework settings. The configuration files provide a centralized mechanism for setting up the information needed by different components of the framework at once.

The framework relies on the following main configuration files:

- **main.conf**
  This file is used to configure general settings for the framework, such as the authentication method, the time gap between each two subsequent collections, the framework log/debug files, etc. The file structure of *main.conf* is shown in Figure 3.

```
[main]
#Specify the method of authentication that will be used while
#establishing connections with the monitored hosts
#[1=password authentication, 2=public key authentication].
auth_method=2

#Specify the time period in seconds between each two subsequent
#raw data collection processes for each monitored host.
scans_schedule=30

#Specify the full path of the file that will be used by the
#framework for logging information and traces related to the
#functionality of the framework.
logs_file=framework.log
```

Fig. 3. Main configuration file structure

- **hosts.conf**
  This configuration file enables specifying the list of hosts or endpoints to be monitored. In addition, it lets us disable the monitoring of any endpoint without the need to remove the instance information from this file. In this way, any excluded endpoint can be remonitored easily just by enabling the corresponding monitoring option in this file.
- **rawdata.conf**
  All the raw data that should be collected for each endpoint, during the monitoring process, are specified in this file. Moreover, this file enables framework administrators to handle the way in which each raw data is collected without the need to modify the framework source code. You only need to state the name of the script file containing the command or group of commands that will handle the raw data collection.

C. Data collection

The data collection or retrieval process goes through three main steps. First, a secure connection is established with every endpoint under monitoring by the framework. Then the actual retrieval of the raw data starts. Finally, the retrieved data are stored in a storage media and shared with the security scanners integrated with the framework.

1) Endpoints authentication: One role of the controller is establishing secure connections with the monitored endpoints to be able to start the raw data collection process. To establish a connection with any endpoint, the controller must go through an authentication procedure. The framework supports two methods of authentication, namely, password authentication, and public key authentication. The framework administrators can choose which method to be used for the authentication by setting the *auth_method* in the framework configuration file *main.conf*.

2) Raw data collection: One of the contributions in this work is to come up with a flexible and generic framework that can be customized to satisfy the needs of the organization or the company deploying the framework without modifying the source code of the framework. Therefore, the retrieval engine will be implemented to handle the general process of the raw data collection while enabling the end users to write their own code in charge of collecting a new raw data item of interest and integrate the written scripts easily with the framework to be used for collecting the new raw data along with the existing ones.

The general process of raw data collection is depicted in Figure 4. The data collection process for an endpoint starts when the controller establishes a connection with the targeted instance successfully. The data retrieval engine reads the corresponding code files/scripts responsible for collecting the raw data and checks their validity. Then the code is sent via requests to the controller. The controller gets the received code executed on the targeted endpoint and then retrieves the collected data and sends it back to the data retrieval engine. Finally, the data retrieval engine formats the retrieved data in a specific structure and stores it in json files.

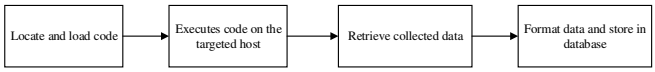

Fig. 4. Data collection procedure

3) Data storing: The data retrieved from the monitored endpoints can be stored in a storage media (e.g., log files, light database, etc.). A database such as MongoDB [1] can be used for this purpose. However, in our implementation of the framework prototype, we will use JSON files for storing the collected data from the monitored endpoints. JSON files will be used for the fact that it allows storing data in a standard data interchange format that is lightweight.

D. Collected raw data

This section explores which type of data can be collected through the framework architecture. No much focus will be paid at this stage on how the data will be processed on the framework side. We would explore, in particular, what can be collected related to the following types of data:

1) **User activity data**
   Examples of user activity data that can be collected through the framework architecture include last logon, failed login attempts, user password last set, group memberships for a user, list of users logged on and the session information for each user, list of active sessions, etc.
2) **Process table**
   Regarding processes running on the monitored endpoints, the framework can collect the following data:

list of all processes running on the system, list of processes using memory space greater than certain value, history of all processes running (successful) or tried to run (failed) on the system, etc.

3) **File system**
Regarding the operations on the file system of an instance, the framework can collect the following data: the set of all files modified within a specific period of time, the set of all files created within a specific period of time, etc.

4) **Registry**
For operations on Windows registry, the framework is able to extract the following data: values of specific registry keys, export of the whole registry keys (registry operations, such as recently added keys, modified keys, etc., can be figured out indirectly by measuring deviation between two subsequent reads).

5) **Network connections**
The network connections data that can be gathered by the framework include the following: DNS resolver cache data of a monitored endpoint, which stores the IP addresses for the websites recently visited from the endpoint, data related to network connections established to and from each monitored endpoint requires scanning of the inbound and outbound network traffic. The proposed framework is agentless and does not have access to the network interfaces of the monitored endpoints. However, one workaround solution may be through routing specific packets from the monitored hosts to the server running the framework.

6) **Resource consumption data**
The following resource consumption data can be collected by the proposed framework: available physical/virtual memory, total physical/virtual memory, disk usage, CPU usage, etc.

IV. Empirical Evaluation

In this section, we conduct the empirical evaluation of the framework, in terms of security and efficiency.

A. Framework Security Evaluation

In this subsection, the security assumptions of the framework will be presented. Then we will study and examine the security of the proposed framework.

1) Framework functionality: Before starting the security evaluation process of the framework, we need to understand how the framework is supposed to work and what is needed for the framework to work properly.

- The framework is installed to run on a server connected to the network on which the endpoints to be monitored are running.
- The framework has access to the endpoints it is supposed to monitor.
- The server on which the framework is running is considered to be secure, to prevent against various attackers' attempts of manipulating the framework configurations and tampering the collected data.
- The security manager or administrator of the framework is supposed to configure the framework properly by setting the right values for all parameters in the framework configuration files, namely, *main.conf*, *hosts.conf*, and *rawdata.conf*.
- Access to the configuration file *hosts.conf* has to be restricted.
- In case of using "public key authentication" to establish connections with the monitored hosts, the security manager of the framework is supposed to send the public key of each monitored endpoint to the framework server using the proper utility.
- The security manager of the framework is supposed to add the required IP address of each new endpoint to the configuration file *hosts.conf*, and maintain this file up to date thoroughly. However, the framework can be easily updated to detect the new endpoints connected to the network.

2) Attacker model: Following the Dolev-Yao threat model [7], any violation of the framework operation is considered as an attack. The attacker is able to intercept, overhear, listen, and modify data exchanged between the framework server and the endpoints. The power of the attacker is only limited by restrictions imposed by the employed standard cryptographic protocols, and the security mechanisms adopted on the framework server. The assumptions considered for the attacker model are as follows:

- The attacker can sniff the network traffic between the framework server and each monitored endpoint.
- The attacker can intercept and manipulate the network packets exchanged between the framework server and each monitored endpoint.
- The attacker can capture the authentication related network traffic and the network traffic of the data collection requests sent from the framework server to each monitored host, and record the corresponding data for later use. For example, to conduct replay attacks.
- A monitored host may get compromised and controlled by the attacker.
- The attacker can spoof the IP address of any of the monitored endpoints.
- The attacker can send connection requests to the framework server.

3) Informal security analysis: Having set up all the machinery of the framework as stated in Section IV-A.1, and given the attacker model described in Section IV-A.2, this section discusses potential attacks that could violate the security properties of the framework, and how the proposed framework design protects against such attacks.

Credentials sniffing attack: Because the attacker has the capability to eavesdrop on the network traffic established during the authentication with any monitored endpoint, he may learn the credentials of the endpoints, for example, and then use them to get control over the monitored endpoints. However, the proposed framework protects against this threat

by, first, using standard and secure protocols to establish connections in a secure way (e.g., SSH). Second, the framework provides another level of security against this attack by using "public key authentication" to establish connections.

Replay attacks: The attacker may use the recorded packets to do several actions, such as establishing connections with the monitored hosts, gathering various data, disturbing the functionality of the targeted endpoints, etc. However, the framework depends on using standard and secure protocols (e.g., SSH) that are proved to protect against replay attacks in general.

Spoofing attack: As discussed in the attacker model, the attacker can spoof the IP address of any monitored endpoint. In this case, the framework will not be able to establish connection with the attacker host as the attacker has to create a user account with the same credentials used by the host with the spoofed IP. In addition, the framework raises an alert when it fails to establish connection with any endpoint. Even in the case where the attacker was able to guess the credentials, the attacker will not be able to do any malicious activity against the framework.

Man-in-the-Middle attack: If the attacker was able to inject himself between the framework and a monitored endpoint, he will be able to either sniff the exchanged traffic and hence he will learn nothing from the encrypted traffic, or he will try to tamper the transferred data and in this case the tampered packets will be discarded. The attacker may use this attack to prevent monitoring of a specific endpoint by keeping tampering the collected data sent back to the framework; however, the framework raises an alert if no data are received from any endpoint for a predefined period of time.

DoS attacks: DoS attacks can be launched by the attacker against the server running the framework to impact the functionality of the service provided by the framework. However, the proposed framework enables the framework administrators to set up thresholds for the minimum required memory and disk space, and alarms will be raised if the available memory or disk space go below those specified thresholds.

Compromising a monitored endpoint: If any of the monitored endpoints got compromised by the attacker and became under his control, the attacker might try to manipulate the source raw data before being collected to mimic a normal behavior and avoids being detected. However, it would be difficult for the attacker to manipulate all the data, as the framework works on collecting several sets of data, most of them are generated and manipulated only by the operating system. In addition, the data are collected on a regular basis and stored in a database, and any deviation from the previous records of the data will be detected by the security model.

B. Framework Efficiency Evaluation

As mentioned in Section III, one of the requirements of the proposed framework is to be scalable in a way that enables the monitoring of large number of endpoints while maintaining the scanning performance. In this section, the efficiency of the framework will be examined in terms of resource consumption. To evaluate the performance of the proposed framework, we ran several experiments where the framework has been used to monitor a group of endpoints while measuring the overhead on the framework server in terms of memory and time required to collect data.

1) Experimental Setup: To evaluate the scalability and the performance of the framework, the framework should be used to monitor a large group of instances. However, preparing such testbed is costly and time consuming. As an alternative, we have used the framework to monitor a private cloud consisting of one physical device and 11 virtual machines. All the physical and virtual machines are configured to run on the same LAN network along with the framework server. The physical host and the virtual machine instances run Ubuntu 16.04 LTS. Finally, the framework has been configured to monitor all those machines.

The specifications of the framework server and the monitored physical and virtual hosts are shown in Table I. The table states the memory and the platform of each instance.

TABLE I
HARDWARE AND PLATFORM SPECIFICATIONS

Given Name	Memory	OStype	Platform
Framework server	4GB	64-bit	Win7
Physical host	8GB	64-bit	ubuntu 16.04
Each virtual host	1GB	64-bit	ubuntu 16.04

C. Collected data

In our experiments to evaluate the efficiency of the framework, the framework has been used to collect the following data from the monitored endpoints:

- Successful login attempts made by users and the currently logged-in users.
- Failed login attempts.
- List of all processes running on the system.
- Performance related data of all running processes in the system.
- Available physical/virtual memory.
- Total physical/virtual memory.
- Disk usage.
- CPU usage.

1) Evaluation: We started our experiments by running the framework to monitor one instance, and then measured the resources needed (memory) to run the framework and the time it takes to collect data from the monitored instance. We repeated this experiment five times and then calculated the average for the consumed memory and the elapsed time. After that, we kept adding one more instance to the list of the monitored instances and measuring the memory overhead and the elapsed time. We limited the number of monitored instances to 12 due to the resources needed to run extra virtual instances. The results are presented in Table II.

Figure 5 shows the average time (in seconds) it takes to collect the data from each instance and store the collected data in the corresponding database. The framework took

TABLE II

OVERHEAD IN TERMS OF MEMORY AND DATA COLLECTION TIME

No. Monitored Instances	RAM(K)	Data Collection Time(sec)
1	12,912	0.518
2	13,060	0.61
3	13,068	0.619
4	13,060	0.653
5	13,060	0.667
6	13,072	0.67
7	13,072	0.690
8	13,132	0.688
9	13,184	0.689
10	13,128	0.701
11	13,304	0.704
12	13,312	0.701

about half second to collect the mentioned data from an instance when it was used to monitor just one instance. To test the scalability of the framework, we kept increasing the number of the monitored instances to measure the effect on the performance of the framework. We can see that the average time needed to collect data from each instance was not impacted significantly by increasing the load on the framework by adding more instances to be monitored.

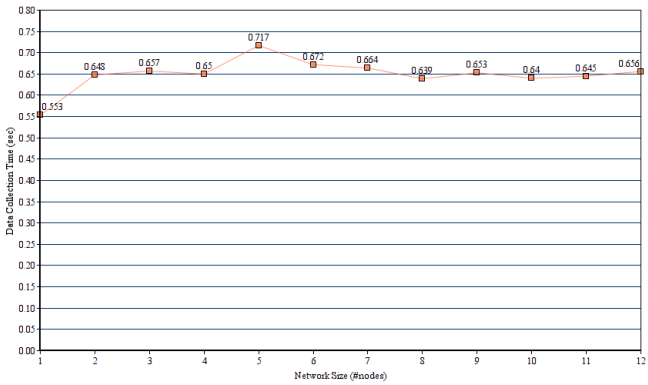

Fig. 5. Data collection time per instance

Figure 6 depicts the amount of memory consumed by the framework on the framework server. The framework needed just about 12 MB of memory. There was only a slight change in the amount of memory consumed by the framework when the number of monitored instances was increased.

V. USE CASE

In this section, we present a use case that shows how the proposed framework can be used to build a real-time intrusion detection model that focuses on detecting abnormal behaviors based on the data collected using the framework.

A. Detection Model

In this use case, a simple intrusion detection model will be presented. The model is for detecting break-ins into computing systems by monitoring users' activities for abnormal patterns. We partially adopt the idea presented in [6]. In the presented model, the behavior of a given user with respect to a given account or machine is captured as an activity profile.

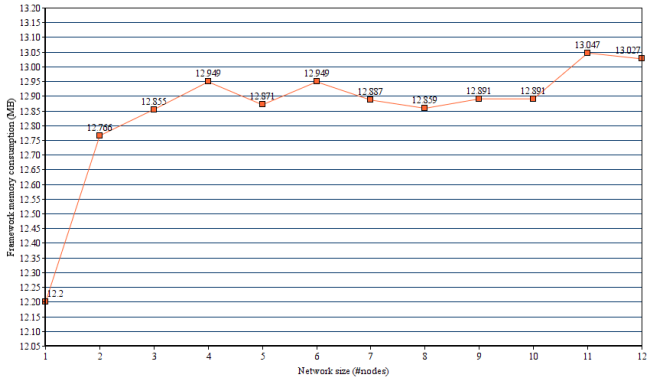

Fig. 6. Framework memory overhead

The activity profile serves as a description or signature of normal activity for its particular user, account or machine. Statistical metrics and models are used to characterize the observed behavior. The focus of the use case will be limited to a detector that monitors and detects attamepted break-in and masquerade attacks.

B. Metrics

In the proposed model, a metric is a representation of a random variable that models accumulated values of quantitative measure over a period of time. We consider the following two types of metrics proposed by Denning [6]:

- **Event counters:** number of events satisfying specific properties.
  Examples: number of successful logins over a time period, number of failed login attempts over a time period.
- **Interval timers:** duration between two related events.
  Examples: time interval between consecutive logins into an account; login interval.

C. Statistical models

To determine whether a new record of specific type of data recently collected using the framework is abnormal, a statistical model is used along with the archived records (past observations) of the same type of data stored in the framework database. The following models suggested by Denning [6] may be considered.

- **Mean and standard deviation model:** computes mean and standard deviation from past observations; if a new observation is not located inside a confidence interval limited by $d \times std$ from the mean, it is considered abnormal. This model can be applied to both event counters and interval timers.
- **Time series model:** an event counter along with an interval timer are used with this model. The model considers the values, the order, and inter-arrival times of a sequence of collected records (observations) $x1, ...xn$ of a data x. If the probability of occurrence of a new collected record of data at that time is too low, it is considered abnormal.

More sophisticated models, e.g., using machine learning, could be considered as well.

D. Profiles

We consider the following profiles as examples:

- **LoginFrequency:** can be represented using an event counter metric which measures the login frequency over a period of time. Unauthorized access to an account by masqueraders can be detected by monitoring login frequencies specifically when the access happens during off-hours in which the account is less likely used by the legitimate user.
- **LastLogin:** can be captured using an interval timer that measures the time elapsed since last login compared against some predefined threshold. Detecting break-in on a dead account, or account with limited activity (for a certain period of time) could be done with the help of this profile.
- **PasswordFails:** captured using an event counter that measures failed password attempts at login compared to some predefined threshold and time window (fairly short one). This can be monitored for individual accounts and for all accounts taken together; so two different thresholds can be defined. This could be useful for detecting break-in attempts.
- **SessionElapsedTime:** can be represented using standard deviation statistical model that measures the elapsed time per session. This model could be useful for detecting masqueraders.

E. Implementation

As a proof of concept, in our initial implementation, we will focus on a subset of the intrusion features as follows:

1) Failed login occurrences: Failed login occurrences over a time interval will lead to two types of events: a warning or an intrusion.

The model analyses the failed login occurrences of a given user with respect to a given account on a machine and generates a warning or intrusion alert. This model takes as input a sequence of timestamps of consecutive failed login attempts and the timestamp of the following successful login, if any.

In case where the number of failed logins over a time interval is above a certain threshold, a warning will be generated indicating a password cracking attempt.

In case where a sequence of failed logins (above a certain threshold) is followed by a successful login over a time interval, a successful break-in through password cracking will be reported as an intrusion.

2) Model building: To build the model, we need first to understand the normal login behavior of a given account over a long period of time and capture that behavior using *PasswordFails* profile as discussed in V-D. Since we are interested here in analyzing the failed logins of a given account, the PasswordFails behavior would be characterized using a time series statistical model and event counter metric, where the number of failed logins for a given user is counted for a time window of 1 minute.

The test environment consists of a server running the framework and another Linux machine used frequently in a real work environment. The framework has been used to monitor the Linux machine for a period of three consecutive work days. The framework has been setup to collect every minute the number of failed login occurrences for a given account, from Linux logs, that happened during the last minute time window. The collected data for the whole period (three days) have been stored in the framework database.

After the successful collection of the failed login attempts data, we have used the collected data to build the *PasswordFails* profile for such account that represents its normal behavior. A visualization of the profile data is shown in Figure 7.

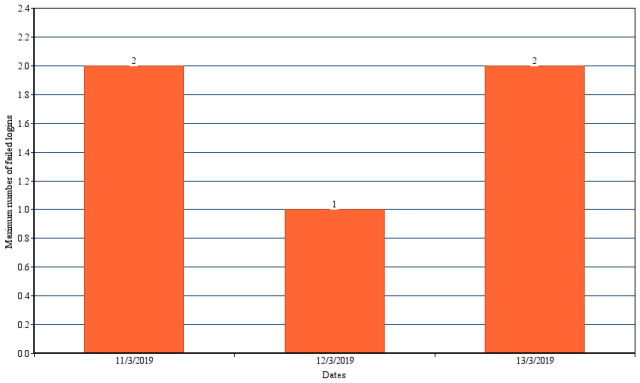

Fig. 7. PasswordFails profile

We can see from the profile that the maximum number of normal failed logins that happened during the three days is two failed login attempts. This number can be used as a threshold in our model to detect attempted break-ins and masqueraders. We have to mention that the normal behavior of failed logins can be more than 2, as we collected the data for only one user. Hence, setting the value of 2 as a threshold may result in a high number of false positives. To reduce the number of false positives in our tests, we will assign the value of 5 to the Threshold parameter. Deciding a value for the threshold when using the developed model in a real world environment would require collecting data for a large group of accounts during a long period of time to build the PasswordFails profile. The data collection scenario proposed here is only to establish a proof of concept. The proposed model has been implemented and run on the framework server. In addition, the model has been integrated with the framework by enabling the model to get access to the collected data stored in the framework database.

3) Model testing: We will conduct two types of tests. The first one is to run the model on the framework server and use the monitored computer for legitimate activities. While in the second test we will use a password cracking tool to crack the password of one of the accounts used to access the monitored endpoint.

Regular login: The purpose of this testing scenario is to check the behavior and the accuracy of the implemented model under normal work scenarios. We have designed three different test cases. The first test case involved providing wrong passwords for 2 times to test if the model would detect any failed logins when their count is under the threshold and also to test for false positives. The second test case involved providing wrong passwords more than 4 times. And the last test case involved providing wrong password 5 times and then providing the correct password to test if the model will detect masqueraders. The results are depicted in Table III.

TABLE III
REGULAR LOGINS TEST CASES

Test Case	Results
Test case 1	None
Test case 2	Failed logins warning
Test case 3	Intrusion alert

Compromised login: In this testing scenario, we will test the ability of the proposed model to detect compromised login attempts when the monitored endpoint undergoes login attacks. We will use **THC-Hydra** password cracking tool to crack the passwords for one of the accounts on the endpoints being monitored by the framework. In this scenario, we will execute Hydra from the attacker machine to conduct password dictionary attack against one of the accounts of the endpoints monitored by the framework. To perform dictionary attack using Hydra, we have created a passwords list of six random passwords that Hydra will try and attempt to access the targeted account. We decided to use a password dictionary of only six passwords rather than conducting brute force attack to test the ability of the implemented model to detect automated attacks when the number of login attempts exceeds the threshold even if the number of failed login attempts is not that large. Linux records all the failed login attempt in the system logs and especially in the file "/var/log/wtmp". The framework reads the data from this file by executing the command *lastb* and provides them to the IDS model.

When we ran the attack command on the attacker node, the IDS model successfully detected the failed logins and generated "Failed login attempts warning." To test the ability of the IDS model on detecting intrusions, we have repeated the same scenario with a slight modification to the passwords list. We added another password to the passwords list which is the correct password for the targeted account. The execution of the attack command resulted in an "Intrusion alert" generated by the IDS model since the failed login attempts were followed by a successful login.

VI. CONCLUSIONS

In this paper, we proposed a software security framework for monitoring cloud computing endpoints—agentlessly—without the need to install any agent on the virtual instances being monitored. Because the existing security mechanisms for monitoring endpoints require installing agents that collect data from the monitored endpoints, they suffer several drawbacks with regards to security, performance, scalability, maintenance and user acceptance, to name a few. We proposed an agentless mechanism that provides the same functionality as the existing agent-based mechanisms while overcoming their drawbacks and challenges. The proposed framework architecture collects data items from the monitored endpoints remotely using standard and secure protocols and stores the collected data in a centralized database that is accessed by various security modules monitoring the security of the endpoints. Moreover, the proposed architecture enables the end users to extend the framework functionality easily, and can be used for monitoring large number of machines. The security of the framework has been evaluated and the results showed the resilience of the framework against potential attacks targeting its security properties. Moreover, experiments showed that increasing the number of monitored instances has very limited effect on the framework performance in collecting the data as well as overhead on the framework server. In the current paper, we have validated the framework by using it to collect a number of raw data that are related to the user activity, file system activity, processes spawned, network traffic, etc. Building on these initial results, in our future work, the framework will be used to collect large scope of data items that are needed for building more sophisticated security models. Much larger scale evaluation of the framework will be conducted, where the framework will be used in large networked environments with endpoints running various platforms (e.g., Linux, Unix, Windows, etc.).

REFERENCES

[1] MongoDB database. https://www.mongodb.com.

[2] What is agentless monitoring? https://www.eginnovations.com/product/agentless-monitoring.

[3] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 35–44. ACM, 2015.

[4] Morgan Brattstrom and Patricia Morreale. Scalable agentless cloud network monitoring. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 171–176. IEEE, 2017.

[5] Jingsong Cui, Hao Xiang, Chi Guo, and Kun Hou. Agentless processes monitoring architecture on cloud platform. In *2014 International Conference on Cloud Computing and Big Data*, pages 1–7. IEEE, 2014.

[6] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

[7] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[8] Hongwei Tang, Shengzhong Feng, Xiaofang Zhao, and Yan Jin. Virtav: An agentless antivirus system based on in-memory signature scanning for virtual machine. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 124–133. IEEE, 2016.