Intel Cloud Integrity Technology 3.0

**Performance Blueprint**

# Background

The central component to Cloud Integrity Technology is the attestation service. This blueprint is the plan to achieve the following goals:

* improve the performance of single and bulk attestation requests
* improve the reliability of the attestation service during peak load times (fewer error responses)
* improve the uptime of the attestation service (fewer memory leaks)
* improve our ability to test the performance of the attestation service for hundreds or thousands of hosts without requiring that amount of server hardware

# Production

## Eliminate Memory Leaks

The attestation service is demonstrating behavior that is limiting its uptime and degrading its performance as the number of records in the database increases.

Work to fix this issue is already in progress and it’s the first priority for improving performance.

## Automatically Refreshing Trust

The attestation service setting “saml.validity.seconds” determines how long a trust assertion is valid after generating it. Each trust assertion is stored in the database so clients can retrieve it.

Because the attestation service knows when each trust assertion will expire, it can run a set of background threads to check for attestations that will expire soon and automatically generate new reports to replace them. The two types of background threads for this activity are a “scheduler” thread and “worker” threads.

The background “scheduler” thread must maintain:

* a list of hosts whose attestations will expire soon (by querying the database periodically)
* a priority queue of remote attestation challenges that should be initiated at the earliest opportunity, where the attestations that expire the earliest are in front
* a priority queue of remote attestation challenges to follow up on, where the challenges with the earliest max ETA are in front

It is the responsibility of the “scheduler” thread to ensure that it doesn’t add any duplicate entries to the queue.

A pool of separate background “worker” threads use the Asynchronous Remote Attestation Challenge process so they can initiate many TPM quotes in a short period of time and follow-up on them later to retrieve the TPM quotes. The “worker” threads will poll the queue for work orders and initiate those remote attestation challenges, pushing a follow-up task containing the host information, nonce, order identifier, and follow-up times (min, max, and average times received from the host added to the current time) received from the host into the follow-up queue. The “worker” threads will not query the database to determine if there is already a current attestation for any host -- they will assume that if an item is in the queue then it needs to be processed.

A client can request the current attestation (default) or a list of all attestations for a host (search) specific attestation report (by id).

When the server handles a request for the current attestation, it must search for the most recent attestation *that is currently valid.*

Example 1. If there is an attestation (A) that is valid since 35 minutes ago thru 25 minutes from now, and a second attestation (B) that is valid since 25 minutes from now thru 1 hour and 25 minutes from now, the correct attestation to return is (A) because it is the most recent attestation *that is currently valid.*  Attestation (B) won’t be valid for another 25 minutes.

Example 2. If there is an attestation (C) that is valid since 1 hour 15 minutes ago thru 15 minutes ago, and no later attestations, then a request for the current attestation must not return any results because there isn’t a current attestation.

Example 3. If there is an attestation (D) that is valid since 45 minutes ago thru 15 minutes from now, and a second attestation (E) that is valid since 5 minutes ago thru 55 minutes from now, the correct attestation to return is (E) because it is the *most recent attestation* *that is currently valid*.

Example 4. If there is an attestation (F) that is valid since 30 minutes ago thru 30 minutes from now, and a second attestation (G) that is valid since 10 minutes ago thru 5 minutes from now, the correct attestation to return is (G) because it is the *most recent attestation that is currently valid.* In this example, the most recent attestation also happens to expire earlier. We still return that one. A second request sent 10 minutes from now (assuming no other attestation reports are generated for that host) would then receive (F) because at that time (G) would have expired so (F) will be the most recent attestation that is currently valid 10 minutes from now.

## Cache Attestation Reports in OpenStack

The OpenStack scheduler checks the fit of each available host for each VM launch requests. In versions 1.x, 2.x, and 3.0 of our OpenStack extensions for attestation this resulted in a query to the attestation service. These queries quickly multiply as the number of concurrent instances launched is increased and the number of available hosts is increased.

The OpenStack scheduler was designed to query its local database so in to improve attestation performance for OpenStack integration, we need to maintain attestation results in the OpenStack database.

The attestation results to store are the overall trust status of the host (trusted or untrusted) and all geotag attributes from the host’s attestation report. The trust and geotag attributes would then be used by the scheduler. Another attribute to store is the expiration time of the trust and geotag attributes.

A background process (or if using ceilometer, an alarm -- see “Opens” below) triggers requests to the attestation service to update the trust status and geotag attributes of each host according to the expiration time.

On the compute nodes, whenever the nova compute service starts it should trigger a new remote attestation by the attestation service. This can be done by updating the expiration time to now (or if using ceilometer, a new measurement can be triggered by events).

This integration assumes the datacenter users consider the OpenStack database to be inside their trust boundary and that nobody will tamper with its contents. In any case, the policy agent may prevent a VM instance with invalid policy from launching and the attestation service will report a VM instance as untrusted if it launched with an invalid policy.

**Opens**: there is a partially completed project in Nova called Extensible Resource Tracking that would be ideal for storing trust and geotag attributes of the host for use in the scheduler (<https://wiki.openstack.org/wiki/ExtensibleResourceTracking> and <https://blueprints.launchpad.net/nova/+spec/extensible-resource-tracking> ) but because it isn’t complete, we may not be able to rely on it immediately - an alternative is to add a meter to the openstack telemetry project (ceilometer) to collect the attestation report for each host (from attestation service) and store it in the telemetry database Reference: <http://docs.openstack.org/developer/ceilometer/> and <http://docs.openstack.org/developer/ceilometer/new_meters.html#add-new-meters> looks like we can add a “Gauge” type meter to measure trust and geotags like “host.trust.status” and “host.tag.attributes”; all the current attributes for a host should be stored as a single “measurement” of dimension “None” so that when they are updated it’s a complete replacement.

## Cache Attestation Reports in Key Broker Proxy

The key broker proxy receives key requests from a compute node, obtains an attestation report for that compute node and forwards it to the key server, then forwards the key server’s response to the compute node.

The key broker proxy can cache the attestation requests locally so that if a valid current attestation report is available, we can skip the request to the attestation service.

The key broker is configurable with the maximum number of seconds that locally cached attestation reports may be valid - individual attestation reports may have an expiration date embedded in the SAML that is earlier, and they would not be reused past that date.

If there is no current and valid locally cached attestation report, the key broker proxy requests a new attestation report for the compute node via a GET request to the attestation service. The attestation service API defines the GET request as only checking the cache and not triggering a new attestation report. Because the attestation service is now Automatically Refreshing Trust, there is a high likelihood that it already has the requested attestation report available for a GET call and will not need to contact the compute node for a remote attestation challenge.

If the key broker proxy does not receive an attestation report from the GET request, then it will send a POST request to the attestation service to force the creation of a new attestation report for that compute node. On the attestation service, this means sending a remote attestation challenge to the compute node, waiting for the response, verifying the response, and generating the SAML trust report.

If the key broker still does not receive an attestation report from a POST request to the attestation service, then it will return an error to its client (the compute node).

## Centralized Task Manager

The attestation service has a few APIs that support bulk operations, and now it also has a background thread to automatically refresh the trust status of hosts. These activities compete for limited resources: JVM threads, database connections, memory, CPU.

If the attestation service is tuned by the administrator to use 25 threads for bulk attestation requests, and 20 concurrent bulk attestation requests are sent to the service, we want to avoid a situation where the service attempts to start 500 new threads to handle these requests.

There should be a centralized task manager that can be configured with maximum allowed threads, and this task manager should receive and queue all work requests, and process them as quickly as possible using all available threads up to the configured maximum. Now if the administrator tunes the service to use 25 threads for bulk attestation, and 20 concurrent bulk attestation requests are sent to the service, the work orders for hosts mentioned in those requests would be added to the queue and dispatched by the centralized task manager to the worker threads.

For asynchronous remote attestations, this is a “fire and forget” operation because the results of the order will be stored in the database and the client will later request the status.

For synchronous remote attestations, the work orders need to maintain a reference to the http thread that submitted them to the task manager, and that http thread needs to sleep until it is notified that the work is done. As each work order is complete, it notifies its http thread and when the http thread notices that all the work orders it dispatched are complete, it sends a response to the client.

When the task manager receives work orders to add to the queue, it should check if there is already a work order in the queue for the same host that can be reused. The two critical parameters of the work order are the host and the nonce. Normally, an attestation request by a client does not specify a nonce and the attestation service generates the nonce for the remote attestation challenge. In these cases, the work order should not have a nonce set because it will be generated when the work order is processed. Multiple concurrent requests that do not specify a nonce can be consolidated: for asynchronous requests, the task manager adds the request id to the work order already in the queue so it will update that request also when it’s done, and for synchronous requests the task manager adds the http thread that submitted the new task to the existing task so it will update that thread also when it’s done. This means the request id and http thread references in each work order must be a list, not a single item, and they should both be initialized to empty list when the work order is created so it’s safe to add elements without checking for null.

The centralized task manager should handle both incoming attestation requests and the background attestation requests created by Automatically Refreshing Trust.

## Asynchronous Remote Attestation Challenge

The Trust Agent version 1.x, 2.x, and 3.0 includes a remote attestation API where the challenger submits a nonce and the trust agent responds with the TPM quote. This is a synchronous procedure - the client must wait for the response, which may take 3 seconds or more depending on the TPM hardware.

A new asynchronous remote attestation API must be added to the Trust Agent that will allow a client to submit a nonce and receive an immediate response with an order identifier and URL where the client can check the order status and a time estimate. The Trust Agent will process the request and store the TPM quote response on disk. The client can then retrieve the response at a later time using the order identifier generated in the first transaction.

The trust agent startup procedure should include a new task that measures the time it takes to generate a TPM quote on that host. That measurement and a measurement count (starting at 1) should be retained in memory. The Trust Agent should maintain the shortest TPM quote time (min), longest TPM quote time (max), and the average TPM quote time (over the number of TPM quotes generated since the Trust Agent was started). Each time a TPM quote is performed, the measurement count should be incremented and the min, max, and average TPM quote response times should be updated. Each time the Trust Agent responds to a new asynchronous request with a new order identifier, it should also include the min, max, and average TPM quote times with the response. This will hint to clients the timeframe that they should schedule their second request to retrieve the TPM quote.

The attestation service versions 1.x, 2.x, and 3.0 have a synchronous attestation API. A new API for asynchronous requests should be added to the attestation service which would work like the trust agent asynchronous remote attestation described above: clients submit requests and get an immediate response with an order id, URL to check order status, and time estimate, then they request a status for that order at a later time and continue requesting status updates until all the work is done. The status updates would show what work is done already in case some of it is done and the rest is pending, so the client can start to act on that data if possible -- for example a UI could display the appropriate icons corresponding to the trust reports that are done while displaying waiting indicators for the ones that are not done yet.

## Caching Authorization Data in Memory

The attestation service maintains a list of users, roles, and permissions. This list changes infrequently compared to whitelists and host registrations and attestation reports.

In attestation service versions 1.x, 2.x, and 3.0, every request to the attestation service is authenticated and this includes one or more database lookups for the user login credentials, the user record, the roles, and permissions.

The response time for every request can be improved by eliminating some of those database lookups.

The attestation service should maintain a limited-size (configurable) cache of recently used security information and check this cache first for credentials before checking the database.

This will eliminate authorization-related database queries for normal users and improve the response time for each request after their first login since the service was started. The cache will not, by itself, improve response time for clients that submit incorrect credentials, as these will not be cached and each will result in a single query to the database to look up the password or certificate login record.

The APIs that edit user and role information in the database will need to also remove the corresponding records from the cache so the next query with those credentials will result in a query to the database for updated information.

## Switch from Tomcat to Jetty and/or Web Server Tuning

The attestation service versions 1.x, 2.x, and 3.0 run on Tomcat and the slow startup time is very prominent during installation. In comparison, the key broker and trust director include Jetty, an embedded web server, and their installation and startup times are faster.

We need to do some performance testing to compare the same applications on both Tomcat and Jetty. It would be easier to create a .war file with key broker or trust director and deploy that on Tomcat to compare that performance with the current Jetty configuration for those applications. Both Tomcat and Jetty should be tuned for the best possible performance for the applications during the test. If Jetty performs better, then we should repackage the attestation service using Jetty the same way key broker and trust director are packaged.

# Testing

We have a limited hardware available for testing the performance of the attestation service, but we need to test it with much higher numbers of hosts.

We can develop simulations for the trust agent and vcenter responses that we can use for testing performance of the attestation service, substituting VMs for hosts.

The /etc/hosts file on the attestation service can be configured to have many host names pointing to the same simulator IP address. Each simulator can then be scaled vertically to handle more simulated hosts until capacity, which is expected to be in the hundreds of simulated hosts per simulator VM, and then horizontally by adding more simulator VMs and corresponding entries in the /etc/hosts file until we have enough simulated hosts for the performance test.

## Allowing Challenger to Specify Nonce

This is a feature that we should have available anyway for audit purposes, as it allows an auditor to verify the chain of trust all the way to the TPM quote for its own request.

For performance testing, it allows us to send the same nonce for all the requests, to allow of responses from the simulators.

This feature adds an optional parameter to the attestation request which is the nonce to use in the remote attestation challenge. If the request does not specify a nonce, the attestation service would generate one as it does in versions 1.x, 2.x, and 3.0.

## Trust Agent Simulator

The trust agent performance can be profiled by sending a remote attestation request to a real trust agent, recording the response. This can be repeated a number of times to get the minimum, maximum, and average response times.

The trust agent simulator can be built by taking a single sample response from the real trust agent and creating a lightweight web server that accepts a request, waits a random amount of time in accordance with the profile, then returns the recorded sample response. Because the recorded response was built with a specific nonce, it will only be “valid” if that same nonce is used to request it via the attestation service feature Allowing Challenger to Specify Nonce.

All simulated trust agent hosts would have the same whitelist.

## VCenter Simulator

The vcenter performance can be profiled by sending a remote attestation request to a real ESXi host via VCenter, recording the response. This can be repeated a number of times to get the minimum, maximum, and average response times.

The trust agent simulator can be built by taking a single sample response from the real trust agent and creating a lightweight web server that accepts a request, waits a random amount of time in accordance with the profile, then returns the recorded sample response. Because VCenter does not accept a nonce for the trust report, the only variable is the hostname. Therefore a single VCenter simulator can simulate many hosts by dynamically inserting the requested hostname into its recorded response.

All simulated ESXi hosts would have the same whitelist.