Integration testing among enterprise security products

Jeyasekar Marimuthu

Jeyasekar.m@gmail.com

Abstract

An ecosystem that consists of millions of managed nodes or internet of things (IoT) that communicate with each other will require diverse integration testing before the system is shipped to customers. The products and nodes that interact with each other are truly heterogeneous. They have few things in common to communicate between them.

Writing integration tests between products of heterogeneous nature needs a modularized approach that can isolate technologies, platforms the products run on. The runtime objects will have many uncommon properties between each other. An integration test framework should be able to handle the uncommon characteristics and validate the integration tests.

This paper highlights the challenges faced in such environments and some of the integration techniques developed to address those challenges. One of the testing techniques is to have developed an integration interface that hides the underlying native tests run on individual system and run the integration tests seamlessly across the enterprise. Some of the tests are SDK based and some are REST based. The paper also reviews some of the tools and techniques used in complex environments. Adapting to development process like Behavior Driven Development (BDD) saves significant amount of time to setup complex environments.

Biography

Jeyasekar Marimuthu is a lead software developer at Intel Security, Hillsboro with comprehensive experience in architecting and developing solutions, project execution, process improvement, cross product development, and client relations. His strengths are: understanding business requirements, evaluating risks, providing architecture and designs, and strategizing profitable execution. A skilled developer and technical leader, Jeyasekar communicates with management, vendors, team members, and staff at all skill levels. He brings innovation with ideas and drives improvements in development and processes. He has 15 years of development experience in enterprise and security platforms using Java, J2EE, database technologies.

.

1 Introduction

An enterprise that wants to protect its assets (work stations, servers and each device attached to the network) from security threats will have to depend on a number of interconnected security solutions across the eco-system.

In such environments, a large enterprise will normally consist of millions of nodes or assets that exchange data are unified through a common data exchange framework. The node data will be processed by each point product and will have to exchange with other products. The verification tests and technologies used to write the tests need not be same between one to other.

Before each point product is readied for integration tests, it is tested with a number of feature verification tests. When there is a need to exchange data among two or more products, the system as a whole needs to be tested with integration tests that can ensure integrity of data.

A key concern when writing integration tests is that each point product deals with a different test framework. The underlying technology between products is not the same. The datatypes used among products to exchange info are heterogeneous in nature. The runtime objects are complex. This paper explains how to address incompatibilities between products and how to perform integration tests seamlessly across products.

Let us look at how this might work.

2 Background

A security ecosystem within IOT (The Internet of Things) is the network of a wide array of physical objects embedded with electronics, software, sensors that offers connectivity to achieve greater value and service. An Ecosystem consists of one or more security products managing data from many (“millions”) of nodes across the network exchanging data between them. A large volume of data is sent to enterprise products through data channels. An enterprise might deploy a number of security products together. Each one of them will have its own kind of tests developed by its respective engineering team. Each security service should be testable under its own test framework. Tests that qualify each point product to production-ready are FVTs (Functional Verification Vests).

The next phase of testing is to integrate all of point products and to perform an integration testing. When the products are unified and deployed in a common operating environment, each set of tests written for system ‘A’ need not be compatible with system ‘B’ due to various reasons. The reasons could be due to less common qualities between systems, or the scripting language used by system ‘A’ may be different from system ‘B’, or setting up an environment with all products may be a challenging through automation.

Any compatibility issue mentioned above will challenge writing and executing system tests. There is a need to address the incompatibilities between products and a need to develop an interface that can execute FVTs and system tests seamlessly. Without such an interface, it will be next to impossible to convert the test object of one system into another using a different scripting language.

3 Enterprise Ecosystem Architecture

Each point product may host one or more security services. A node in an enterprise network can leverage service from one or more security products. The data exchanged between subsystems or point products needs validation before a point product sends a request to use services from other point products. The

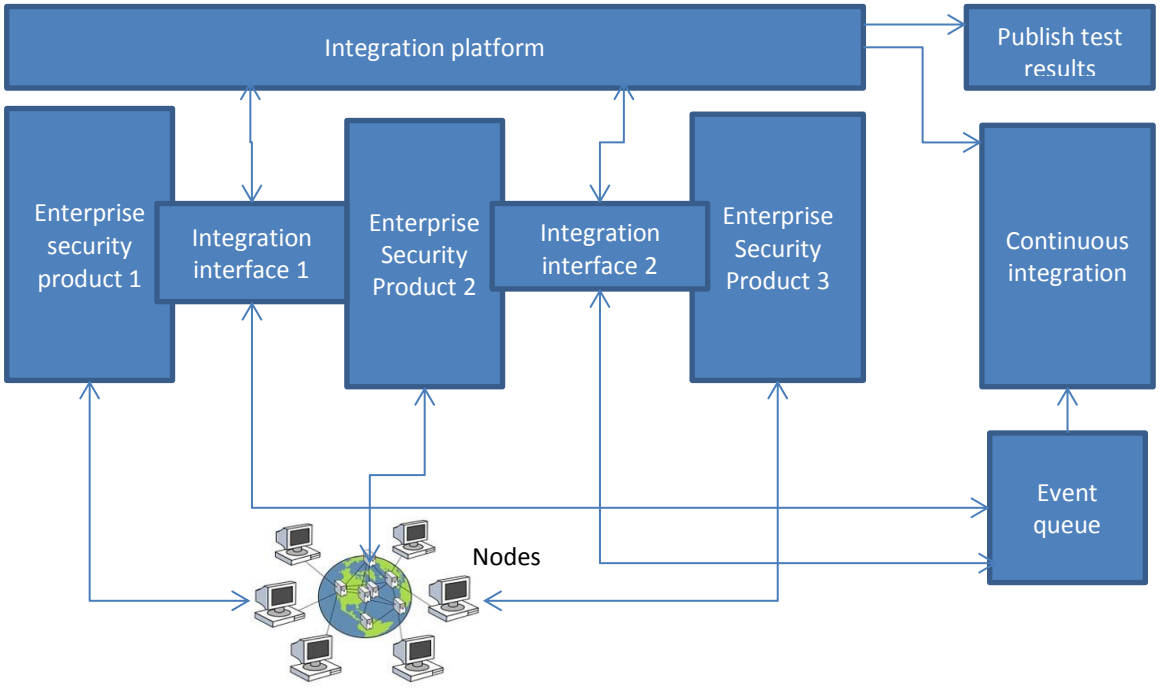
following sequence of automation ensures the testability of a point product in standalone mode.

1. Ensure each service or function point has sufficient unit tests.
2. BVTs (Build Verification Tests) are present.

3. Whenever there are build changes BVTs are run by a continuous integration server like Teamcity.

The following diagram explains how an integration test platform should be. The integration framework consists of the following components.

1. Nodes or assets that are linked to the enterprise network. Each network may have a number of subnets.
2. One or more security products that consume the data collected from loads of assets or nodes.
3. An interface that bridges one product with other. Depending on the number of interconnections, the number of interfaces will vary.
4. An integration platform that contains platform independent abstract definitions for various interfaces.
5. An event queue that aggregates all the runtime test results and queues them as events.
6. A continuous integration server that consumes the events from event queue and runs the integration tests.
7. A management console that publishes the integration test results.



4 Integration Tests In Security Ecosystems

Integration testing (sometimes called integration and testing) is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing. The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. Carrying out integration tests on an enterprise ecosystem that contains two or more point products and exchanges data between them must consider the following system design.

1. Each subsystem or point product is unique in nature
2. Unit tests of each subsystem might not have been written in the same language in which the system tests are created.
3. Understand the platform specific tests and use them for the system automation framework.
4. Data that come in from various products in asynchronous fashion.
5. Each subsystem deals with a number of client machines.
6. Each client can subscribe to one or more services across the network.

Integration between subsystems normally happens in the following way.

1. Feature Verification Tests are run on each subsystem before the results are sent for system tests.
2. Integration tests are written. Each system test deals with source and target objects.
3. The system deals with a definite size of data for tests.
4. An automated or manual system to review the results and to correct if there are issues.
5. Integration tests are executed on need basis.

5 Integration Automation Tests and Challenges

System automation tests written for one security system may not be runnable on another system due to their heterogeneous nature. Simulation of a system and mocking the behavior is key to automating system tests. Upon successful mocking, the test system should make sure the tests are run seamlessly from system to system by dealing with the technology crossovers.

Some of the key challenges are listed here.

1. Integration tests are written and each integration system deals with logic that converts the original call into a native call that the targeted system can understand. This consumes a lot of time to complete all integration tests. A chunk of redundant code exists in each integration test.
2. Tight coupling between the source and target systems.
3. Handling varying data size from time to time and lack of performance loops in the automation cycle.
4. Executing integration tests on need basis. They are not continuously built within build room.
5. No coordination among point products builds. The FVT and integration cycles are independent in nature.
6. Adapting to tests that are written using unfamiliar scripting language and the time to learn the targeted platform and to convert tests for the same.
7. A significant learning curve to mock objects for targeted systems.
8. Challenges in setting up the environment by automation scripts.

6 How to fix?

This model simplifies integration automation testing.

1. We do not intend to learn and write tests for each subsystem.
2. We do not intend to change the underlying language that the system tests are written in.
3. The system tests should make use of the existing unit tests from each subsystem.
4. The integration system should be horizontally scalable as it needs to deal with more security services.
5. Results obtained from subsystems should be unifiable and should produce one collective report
6. Tests should be integrated into a continuous integration build system.

The solution to this scenario is as follows.

Write an adapter for each subsystem that can receive instructions from the system automation framework and can convert them into native subsystem calls.

1. Use Behavior Driven Development (BDD) for setting up environments. This reduces the learning curve for automation engineers. It allows setting up multiple test environments with multiple configurations. It simplifies traceability between stories and test scenarios.
2. Use BDD for feature injection.
3. Define request and response objects that follow adapter patterns and work bidirectional.
4. Create mock objects according to the generic object model and let the adapter convert them into the targeted system object in runtime. For example, Source system A has a number of Python tests and Target B has a number of Java tests. Writing an adapter that accepts python calls and invoke a java tests with appropriate arguments saves the integration tester's effort significantly. When the adapter is prototyped, it makes anyone job further easy.
5. Achieve seamless integration into any number of subsystem by just writing an adapter rather writing a number of native redundant tests.
6. Feed the results into the performance test system. This paper does not focus on accomplishing performance tests.
7. Integrate tests into continuous integration loops using build servers like Teamcity. The Teamcity builds will do the following.
    a. Run FVTs on each subsystem nightly
    b. Create runtime environments using BDD (say using Gherkin)
    c. Deploy builds on the virtual environment and trigger system tests
    d. The Adapters in the integration framework invoke the appropriate native tests as part of the Teamcity build.
    e. Publish results from the build system.

7 Advantages

1. No expertise needed to configure environments: When BDD is used, the framework does not demand every tester to write the underlying scripts.
2. Seamless integration: When an interface is written to integrate any source and destination system, the integration tester is going to focus only on writing in the language that the system is built with. He/she can just plug and play the tests for the target platform.
3. Cost saving: When there is time saving, it reflects in costs as well.
4. Scalable: As there could be more tests added dynamically using integration framework, addition of tests prove to be scalable.
5. Focus remains only on writing integration tests. Configuring and learning underlying technologies are not required.

8 Conclusion

Developing an integration framework with one or more interfaces can seamlessly deal with multiple security point products. When the complexity of environments grow, it is recommended to go with tools that use Behavior Driven Development. It can simplify the task of writing many lines of code. Also, it allows creating multiple virtual environments with diverse settings as the products require.

The defined model works very well for enterprise setups that deal with few million nodes. When the network is expanded with more IOT devices, the heterogeneous nature of systems and the size of data will expand unimaginably. In order to address such needs, the integration framework will need to be revisited according to growing needs.

References

- https://en.wikipedia.org/wiki/Internet_of_Things

"Internet of Things." *Wikipedia*. Wikimedia Foundation, 03 Aug. 2015.

- https://en.wikipedia.org/wiki/Integration_testing

"Integration Testing." *Wikipedia*. Wikimedia Foundation, 26 May. 2015.

- https://pythonhosted.org/behave/philosophy.html

"Behavior Driven Development." *Behave 1.2.5 Documentation*. 07 Aug. 2015.