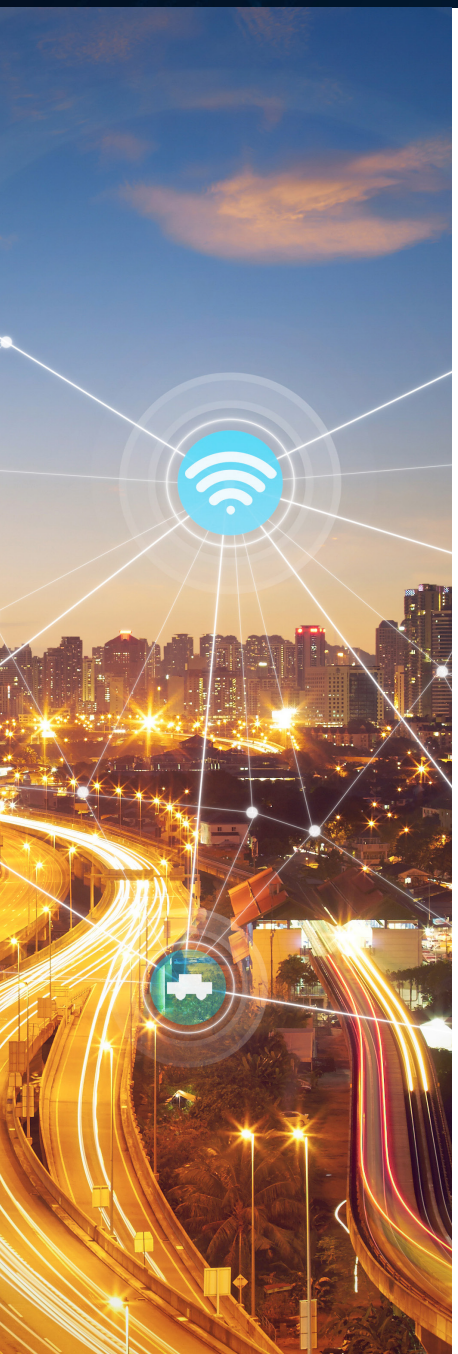
End-to-End Testing for IoT Integrity

EXECUTIVE SUMMARY

IoT solutions are composed of at least two layers, often a device at the front-end that collects data and performs specific actions, and a back-end application that processes the data and provides commands. But even in very simple IoT solutions, it can be difficult to ensure that end-to-end data and control flow are secure, reliable, and compliant due to the blend of disparate technologies, such as low-level microcontrollers and higher-level server programming. In this paper, we'll discuss the testing challenges associated with IoT architectures, including the effectiveness of an end-to-end approach to IoT system verification.



WHY THE EVOLUTION OF IOT PRESENTS TESTING CHALLENGES

"Internet of Things" solutions are evolving, and testing them requires a comprehensive thought process. Although testing practices and activities are well known, the density and the combination of technologies are changing. In the past, embedded software engineering was positioned apart from other areas of the software development landscape, but this is no longer the case, as the entire stack of applications can often function as a single solution. People in charge of system quality need to understand the challenges related to low-level embedded testing, as well as the challenges related to service components.

The structure of IoT systems has evolved beyond simple client devices providing data or receiving instructions. Instead of simply performing actions based on instructions received from "the cloud," IoT systems now also make decisions based on data from within their subsystem. The idea that IoT components may be connected in complex configurations requires us to think about the exponentially expanding problem of testing IoT architectures.

When you create applications that publish and/or consume services within an IoT environment, it is your responsibility to ensure that all elements of the solution stack function correctly. This usually includes low-level testing of the microcontroller layer, as well as higher-level verification of transactions between the various endpoints. If your application fails to deliver the expected results, your customers and partners will not care whether the failure stems from code you developed or from a component you've integrated.

The wide array of technologies now being deployed in very basic solutions used to be characteristic of bigger, more expensive projects, such as logistic systems developed for large organizations, which often rely on homegrown solutions for testing. In the current IoT world, knowledge of different technologies is important, but assuring that the information is correctly exchanged and interpreted from end-to-end is equally important. Organizations should be equipped with easy-to-use and reliable testing solutions that are able to test such a broad array of technologies and help analyze testing results from different sub-components of the solution.

PARASOFT®
Automated Software Testing

EXAMPLE IOT SOLUTION

We'll frame our discussion about end-to-end testing for IoT systems within the context of the following example. Consider a medical solution that consist of the following components:

- Wearable blood sensor (e.g., glucose sensor)
- Wearable medication injector (e.g., insulin)
- Smartphone
- Cloud-based healthcare system

In this example, a sensor monitors the blood for a certain parameter, such as glucose level, and performs measurements in timed intervals. The intervals can be changed with a dedicated command, while another command performs measurements on demand. Measurements are sent wirelessly to an app on the smartphone. The app stores the values locally in a database and provides an interface so that the user can observe the changes of glucose levels in the blood, monitor and compare historical data, monitor changes around meal times, and take other actions. The insulin injector follows a programmed schedule, but if it's disconnected from the other components of the system (e.g., disruption to the network), then it can follow a "safe" plan of injections.

The smartphone app serves as a middleware, and in addition to providing simple analysis to the user. The app forwards glucose level information to a cloud-based healthcare system for additional processing. The cloud application compares the current measurement to historical data and performs advanced analysis, looking for unwanted patterns. In the case of potential danger, the system sends a warning directly to the user or reports to a human medical staff. A specialized medical consultant can immediately analyze the data and decide whether or not a special alert to the patient is appropriate.

The patient receives an alarm and guidance for the next steps, which can include changes in the medication dosing. After the patient's approval, changes to the medication dose are delivered to injector. If the patient does not respond, then the medical consultant can use the GPS information packaged with the smartphone data to send an ambulance.

From the technology perspective, we assume that the glucose sensor and injector are MCU-based systems developed in C language. The middleware application in the smartphone and the cloud back-end are developed in Java. The sensor, smartphone app, and injector exchange the data wirelessly using a custom protocol. The smartphone app and cloud healthcare system exchange data using HTTP.

THE EFFECTIVENESS OF END-TO-END TESTING FOR IOT

Very often in many organizations, end-to-end testing performed at the system level is the first and only choice, mainly because (on the surface) fully assembling the system and testing it seems like the most logical and effective use of QA resources. Testing user requirements on the integrated system would seem to be the closest the organization would get to realistically simulating the final end user experience. But while end-to-end testing should be a part of your QA activities, relying on it as the primary QA practice often leads to late-cycle detection of errors, and delays in releasing products to market.

To be clear, end-to-end testing is valuable and should be done—but it should be used to ensure that the pieces fit together properly, not as a means to detect defects. The system will be able to validate some requirements, but it cannot simulate all types of situations, and building a system that can do this is costly and time-consuming.

Testing the example system described above poses a significant challenge because it is comprised of several separate applications working in tandem. When performing test scenarios, simulating an acceptable range of interactions at different ends of the system may be difficult. The same applies to response verification. In this example, the primary test case may include the following actions:

1. Stimulating the blood sensor
2. Inducing a data package
3. Propagating the data to the cloud
4. Generating an alert from the cloud
5. Waiting for response from human medical staff
6. Generating alarm/notification for the patient
7. Changing the injection schedule or delivering an immediate injection

Later-stage testing to verify this scenario is extremely labor intensive and costly. Additionally, as more organizations move toward agile development approaches, it becomes more and more difficult to

adequately test within iterations. There are also potentially long waiting periods during the development process to start testing. Organizations are faced with a trade-off decision between speed and quality.

Automated system tests can address some of these issues, but automation at this high level is very time consuming and unreliable. Additionally, issues with test-infrastructure access result in development teams becoming overwhelmed with problem reports in between long periods of silence from QA.

In order to test this scenario and find potential root causes of problems that may emerge, components should be isolated and tested early. By shifting testing to the left, organizations can leverage more effective automation to find and fix systemic defects early in the process. Software engineers and testers can generate more useful data about the code to prioritize tasks according to their development policy, saving time and money over the entire development lifecycle. These are well-known concepts in software testing, gaining importance in IoT world.

Deconstructing the System into Layers for More Effective Testing

There are two primary challenges associated with decomposing applications into their constituent layers. **First**, designing the solution in a way that is conducive to segmenting it into smaller building blocks with well-defined interfaces. **Second**, building automated testing frameworks around these smaller units. It is up to the organization to decide on the level of granularity for testing. Depending on the application, you might test at the function level with pure unit testing, or target the integration level, for instance.

In general, a test plan should include a combination of unit testing, integration testing, and end-to-end testing. The proportion of unit tests to integration tests may vary depending on the complexity of the solution. The more complex the solution, the more important unit tests become, because as the complexity of the software grows, it becomes more difficult to stimulate high-level interfaces that ensure that the various paths are executed.

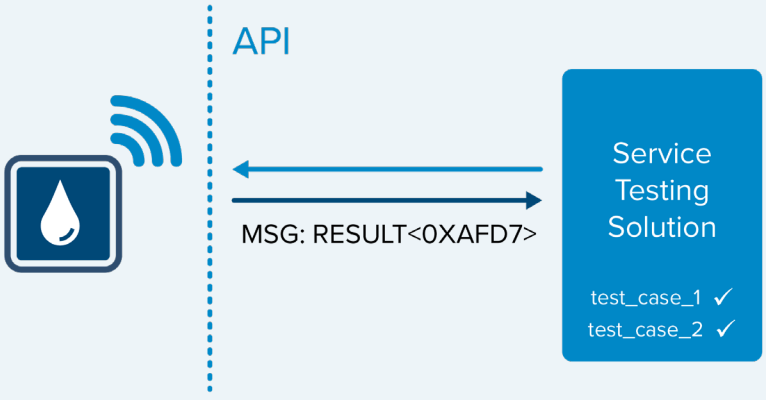
Unit testing, though, is expensive in terms of time and resources. Someone with programming expertise, i.e., an engineer, must write the tests. And because they are closely tied to the code, unit tests are also brittle. Changes to the code could easily impact tests, so an engineer is also required to constantly maintain the tests. As you move further up the stack, functional-level tests are less prone to breakage, but it becomes harder to identify systemic issues. When a unit test fails, on the other hand, identifying the root causes is easier. This trade-off is why we recommend a blended approach for testing not just IoT environments, but any application where speed and quality matter.

For IoT solutions, the first natural layer contains the wireless communication components. This is where sub-systems interact via APIs. Underneath the APIs are messaging protocols, such as MQTT or HTTP sending payloads, such as JSON or REST, as well as proprietary protocols and binary payloads. In the IoT world, the communication usually follows a publish/subscribe or request/response (e.g. post/get) communication model. The publish/subscribe model involves broadcasting data while other components listen for and consume the published data and perform an action. The request/response model involves sending a message to a service directly and asynchronously waiting for a response.

In any case, the overall approach will be to use the module interface definition to build the test suites and automate the execution. In our medical device example, determining testable subsystems is simple: testing can be focused around the blood sensor, injector, cloud application, and mobile phone application.

AUTOMATING TESTS FOR LOW-LEVEL EMBEDDED SUBCOMPONENTS

The blood sensor and injector are examples of low-level embedded devices that interact with other components of in an IoT solution using services. Because they do not have an API vs a complex user interface, the test automation task is much easier. To do this, we need a framework for stimulating the SUT (system under test) and verify its response, as shown to the right.



For simple scenarios, python or simple scripts can serve as service testing solutions. There are a number of scripting utilities that easily allow you to send simple payloads for testing purposes. But this tactic doesn't scale.

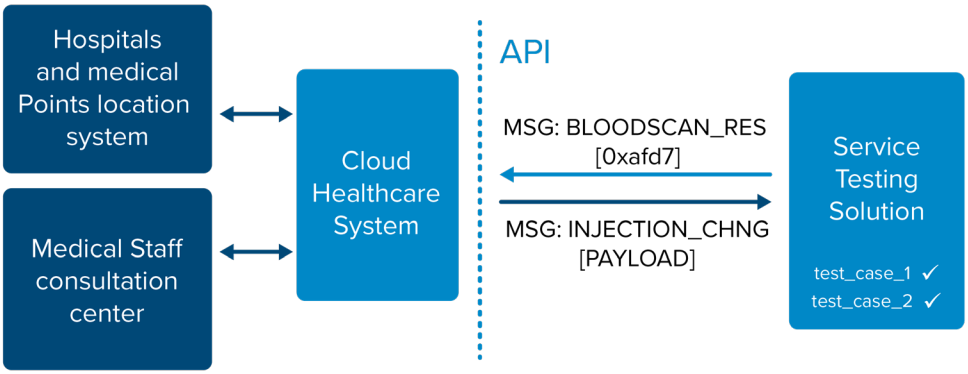
As the number of test scenarios grows, adding and managing the scripts becomes difficult and inefficient. Implementing more complex parameters or sequences is very difficult to do with simple scripting tools. For example, if you want to make a call depending on the value of a previous call, or verify that a particular element is in the payload, you may end up spending a lot of time building your own solution.

Results validation and consistent reporting are also difficult to achieve. A dedicated tool that simulates a range of payloads based on realistic test scenarios ensures the best results.

AUTOMATING TESTS FOR THE SERVER COMPONENTS

On the back-end of the example solution is a cloud-based healthcare system that receives regular reports from each registered patient about glucose levels. The database contains all historical data, as well as additional patient-specific health information.

The process is similar to testing the sensor: simulate the system by sending a simulated package of data from the patient. If a new blood scan shows a significant deviation, the system must determine the appropriate warning to return. High-priority warnings involve human medical staff consultation. The warning message sent back to the patient may contain information about the nearest hospital, urgent care, or other medical facility, as well as a new injection schedule. This is presented on the schema below:



The testing framework will execute a number of test cases containing different values for different patients and expect specific warnings to be generated in response.

But the server isn't just one system – it's a system of systems. And those systems may be unavailable for testing or unable to provide the correct test data necessary to run the test scenarios. These systems don't just provide data, they can also provide logical responses based on the input values. As such, we need to be able to create realistic simulations of what the back-end system is doing. (Moreover, these systems may not even be ready, which is a common challenge in an agile development approach.)

The same issue exists when testing the sensor, which may have dependencies on other subsystems or system components. For example, the blood sensor may need to make a request to another wearable medical device, such as an insulin pump, to verify that injecting glucose is safe. A dedicated service virtualization solution can help by simulating the human response to certain scenarios.

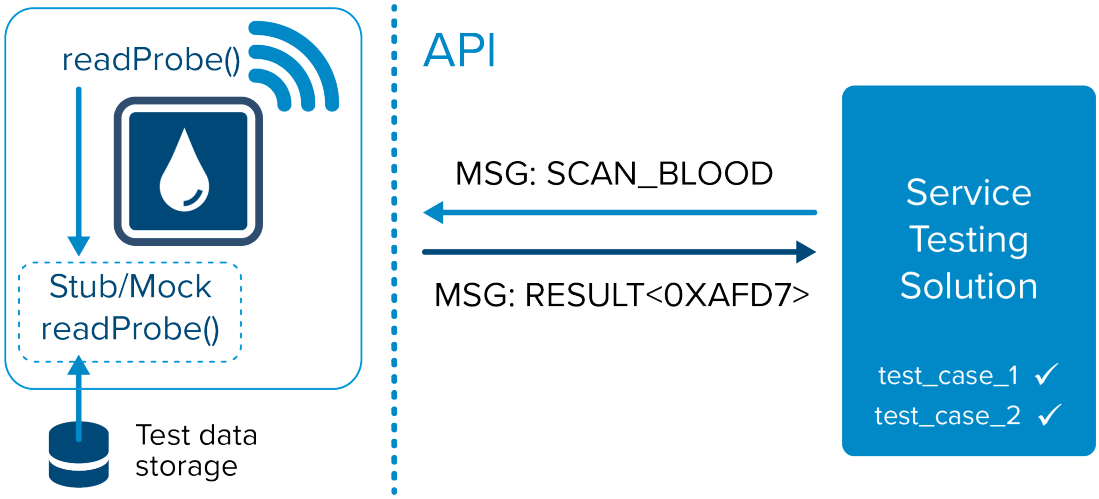
Isolating Components to Improve Automation

In most cases, achieving sufficient automation around the tested components requires the interactions to be isolated from the real world. The strategies for isolation depend on the technology and the type of interactions.

ISOLATING THE LOW-LEVEL EMBEDDED SENSOR

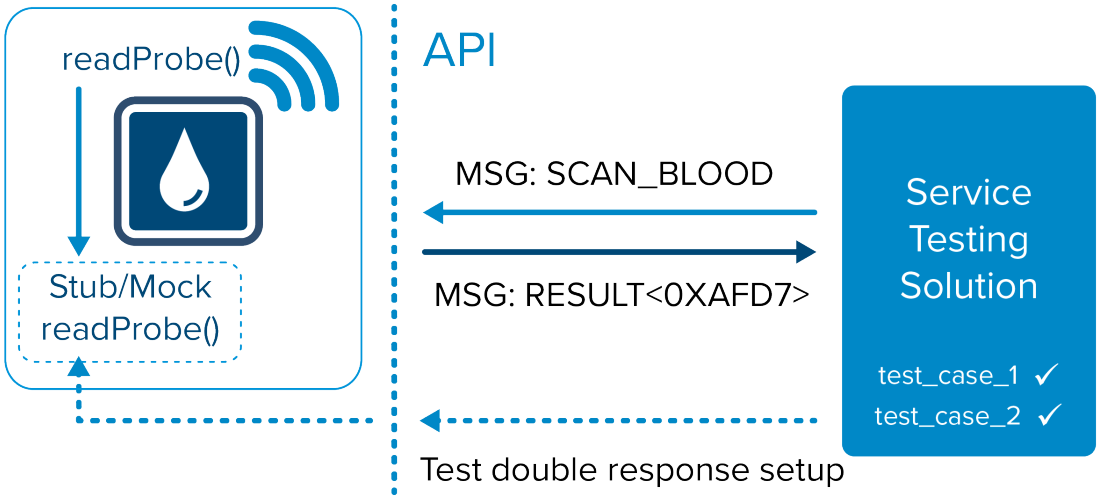
After the SUT is stimulated, it will start performing the requested activity. In this case, scanning the blood to determine the glucose level. The question is how to provide a reliable input to the tested system so that it can perform its activity and respond to the test framework with a result that can be positively validated.

We could potentially place the sensor probe in a liquid with a known level of glucose and other parameters corresponding to real blood. But this approach is neither practical nor scalable. A better approach would be to intercept the function call that reads from the probe and redirect it to a testware stub/mock or virtual asset for generating and simulating a response. This will eliminate the probe itself from the testing process, while enabling the opportunity to perform automated testing of all other parts of the system. The basic idea for this approach is presented below:



A stub or mock (sometimes called a test double) of the interaction needs to be installed to emulate a hardware function call. The stub should be able to respond with reasonable values during the test process using hard-coded values or reading test data from an external data source.

If the test framework requires significant flexibility for the response configuration, a larger investment into the testware may be necessary to provide an API that can be used for setting parameters for the stub before executing the test case:



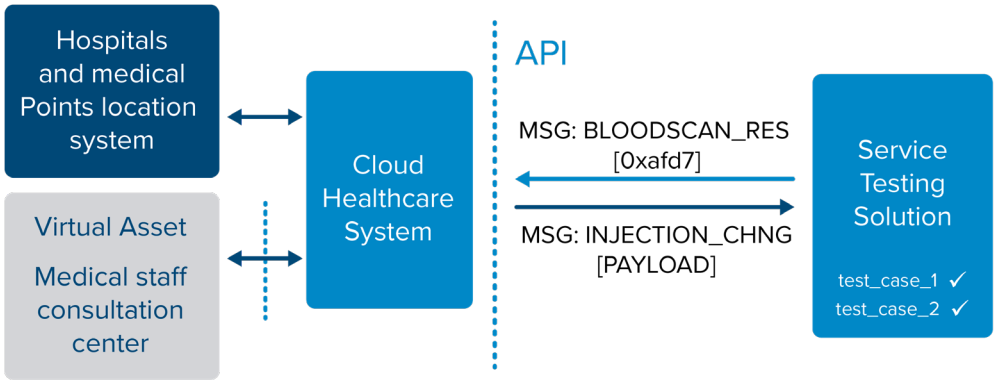
The difficulty of stubbing the tested system depends on a few factors. If its interactions with the outside world are complicated, or if the code was not designed to be testable, implementing stubs to emulate functions from other components will be more difficult. We assume that our example is a simple C language function. The function can easily be emulated with a test stub that can be installed or injected by conditional compilation. With enough forethought, the application can be structured so that functions are appropriately distributed across the source files. This would enable us to swipe out several object files from the linking command line and replace them with object files containing test versions of the functions.

For more complex cases, we recommend using code instrumentation tools, which can automatically stand in place of original definition just before the compilation. Using a code instrumentation tool greatly reduces the effort spent on test asset management.

ISOLATING SERVER COMPONENTS

On the server side of the solution, we have a similar challenge related to test automation.

Part of the tested system is a service that provides human medical consultations. A group of medical experts are connected to the prioritized queue for consultation requests. The first person that becomes available processes the next highest priority request. To enable automated testing, we need to replace this part of the system with a virtual equivalent. Consider the below:



For simple test cases, you can use a stub that contains a node.js script to produce standard responses. But for more advanced scenarios, a service virtualization tool can allow you to define a response depending on the input pattern. You can also use service virtualization to record actual traffic and replay it to simulate the real operation of a connected system. In some cases, you may want to switch between a real system and virtual asset.

Requirements and Code Coverage:

ARE YOU FINISHED TESTING?

A well-designed test framework allows you to easily add test cases and automate execution, but how do you determine if you've done enough testing? You can cover the existing requirements with tests as your baseline.

A common grievance amongst testers, though, is the lack of requirements — especially quality requirements that are conducive to crafting applicable tests. Unfortunately, there are no solutions, automated or otherwise, for producing good functional requirements.

If the requirements are complete and test cases have been created to cover them, in most cases you will gain visibility around the impact of failing test cases. But we also need a way to test non-functional requirements to understand what happens inside the system as the code is exercised. Is the application secure? Does the application perform reliably? Where is the business risk?

We can answer these questions by implementing a tool to capture code coverage, collect metrics, and correlate them with test results to make informed decisions about when to release and the impact of change.

Code coverage tools are often used in conjunction with the execution of unit tests. The code is instrumented and, as the tests execute, data about the parts of the code that are touched is captured. As stated earlier, unit tests are brittle and expensive to implement, which presents a challenge. Additionally, functional testing may also be required to exercise code that's difficult to run with unit tests. Therefore to obtain accurate and complete code coverage information, data from both unit and functional tests needs to be merged, taking into account overlapping coverage. This is very important if the system is a safety-critical application, such as our example, where accurate coverage is necessary for FDA approval.

Some testing solutions on the market are able to collect coverage data from a variety of testing activities and merge them to provide accurate coverage results. The key is collecting both **code coverage**, as it relates to the structure of the physical code, as well as **requirements coverage**, as it relates to the stated and assumed requirements.

With a code coverage tool in place, it is much easier to analyze the quality of test cases. Practice shows that even after executing all test scenarios that seem to cover all requirements, some portions of code may not have been executed. Analyzing uncovered sections of the code allows you to make a decision about whether additional testing is required. It is common that those uncovered sections of the code require special attention — it may be error-handling code or code for processing unexpected inputs. It may appear that the easiest way for verifying if such code works correctly is by applying unit tests together with stubbing/mocking frameworks to override standard behavior of software components and to force code execution to go through rare path.

Finally, it is extremely valuable to tie the code coverage results to the requirements. In this way, we can understand the risk, which may be associated with specific test cases. The coverage information can help you prioritize tests as a result of change.

LOAD TESTING

Verifying the system correctness cannot be accomplished without checking how it responds under extreme conditions, such as heavy traffic. With IoT solutions, not all components have the same performance characteristics or requirements/service level agreements (SLA).

For example, the sensor layer, which is predominantly a data producer, is very unlikely to become a bottleneck and has limited exposure to heavy traffic. The server side, however, should absolutely be verified for handling the defined maximum load.

This goal can be accomplished with dedicated performance testing tools, but the key is to set up and validate the correct SLA at each layer within the application by asking fundamental questions of each layer: How many sensors are connecting back to the server? How frequently are they sending and receiving data? What is the response time from the server to the sensor?

When executing performance testing of an individual component, just as with automated testing under normal working conditions, you need to isolate the component from its dependencies. Service virtualization enables you to simulate different SLA scenarios, such as partial payloads and extreme latencies, to uncover hidden problems and run corner-case scenarios that are impossible to achieve in a full end-to-end test environment.

SECURITY TESTING

Ensuring the security of the solution is one of the most demanding tasks during the development. This is because it relies heavily on the team's experience. It is not only about the QA team maintaining their knowledge of specific threats, though. It is also about the developers being aware of unwanted code patterns that introduce vulnerabilities. Establishing a regular code review process among development and QA teams can help. Developers should review the source code they create and mutually educate them, while QA teams should review created test scenarios to ensure that they build security into the system.

You should automate the execution of source code static analysis tools as part of the development process to expose patterns that lead to security vulnerabilities. A static analysis solution that implements good programming practices, such as CWE or CERT, will scan the code and check for patterns that may lead to exploitable code.

It is especially critical within IoT environments that the static analysis tool can cover all the coding technologies used for our system. Failing to analyze even one small component of a solution without security-oriented scanning may leave an opening for a malicious hacker to infect the entire system.

Executing static analysis should be part of the development policy:

- Define the set of patterns that should be avoided to prevent defects that are a risk to your organization.
- Use early stage flow analysis or dynamic analysis to identify potential weaknesses prior to system tests.
- Use penetration testing to test the APIs.

At each stage, the policy should be refined based on the problems reported. In this way, you are able to systemically prevent defects.

CONCLUSION

Automated software quality activities such as testing of requirements, collecting code coverage, unit testing, or load testing, are not new. They are well-known techniques in software testing, but not necessarily considered when building a "normal" system. In an IoT environment, these techniques must be considered. IoT systems require thinking about software quality in a larger scope.

IoT solutions, such as our medical device example, are different from "normal" systems because an individual feature or function may span multiple layers of the solution. An action initiated at a low-level embedded sensor implemented with C language can trigger messages that are propagated via a middleware broker codded in Java to the back-end, which may be implemented with .NET or Java. All of these layers form a single functional chain, which is only as reliable as the weakest link.

Delivering a high-quality system requires testing capabilities at every layer: the low-level layer in C code, the API or SOA testing layer, and the hard-to-access back-end part of the solution. But it is equally important to equip teams with a reporting framework that allows testing data aggregation from different test activities, so that it is possible to access the quality and completeness of the feature implementation throughout all the layers of the system.

Consider the cost associated with the example system we discussed in this paper. A design failure far outweighs the cost of deploying a testing solution that enables you to isolate and test components, simulate a variety of message payloads with API or SOA testing, and simulate third-party or back-end systems that you may not always have access to.

ABOUT PARASOFT

Parasoft helps organizations perfect today's highly-connected applications by automating time-consuming testing tasks and providing management with intelligent analytics necessary to focus on what matters. Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software, by integrating static and runtime analysis; unit, functional, and API testing; and service virtualization. With developer testing tools, manager reporting/analytics, and executive dashboarding, Parasoft supports software organizations with the innovative tools they need to successfully develop and deploy applications in the embedded, enterprise, and IoT markets, all while enabling today's most strategic development initiatives — agile, continuous testing, DevOps, and security.

www.parasoft.com

Parasoft Headquarters:
+1-626-256-3680

Parasoft EMEA:
+31-70-3922000

Parasoft APAC:
+65-6338-3628

PARASOFT®
Automated Software Testing