# Collaboration in a Secure Development Process Part 3

Gunnar Peterson

## Introduction

In parts one and two, we introduced a hypothetical software development process called the *Secure Development Process*. We assessed strengths and gaps in support of security in standard widely deployed software development methodologies. In part 3 of this series we examine Coding and Deployment Phases in our hypothetical process with an eye on building upon the foundation of analysis and design materials we worked on in earlier phases and towards identifying the critical path to getting secure software into production use.

Note that the activities that we describe in this process can be applied in a top down fashion or incrementally as a grassroots initiative by the development and security teams.

## Activities in the Coding Phase

The coding phase is characterized by building and unit testing code; and refining analysis and design requirements such as architecture and design models which may need to be updated to reflect outcomes of proof of concept work, prototype feedback and changing business requirements and realities.

In many cases on enterprise software development projects, security is not engaged in earnest until the coding or deployment phases. However, we are not starting from scratch here since our stakeholders wisely saw value to have us involved since early on, we the security team have a rich understanding of the application under development, the key players, and a set of artifacts from which to baseline future design decisions.

### Unit Testing

Agile Methodologies such as Extreme Programming (XP) have been criticized from some corners of the software security community for somehow being "bad" for security. While there is relatively less emphasis on design in XP than say Rational Unified Process (RUP), XP has made a very important contributions to code quality including stressing the importance of Unit Testing.

The Unit Test itself is a relatively simple test case (or cases) which execute against the code under development to assess quality and if the code meets requirements. XP introduced a subtle but important point which is to focus on writing test cases before the coding work begins, in other words before the team has consciously begun the process of inevitably making compromises on designs and requirements as development rolls along.

What makes Unit Tests so powerful? The primary issue is that they execute against the code. Excellence in modelling and analysis impacts the chance of successfully making the code iteself

excellent, but the executable code is the final arbiter of its quality. Since Unit Tests work directly with the code they have the leverage to be an important factor in code quality.

The active ingredients in Unit Tests are assertions. Assertions harness the application's methods to discover whether the test case proves true or false. Assertions check the application's actual return against an expected return [1]. From a security standpoint, it can be beneficial to use assertions to check for conditions that should return false, for example checking to see if a middle tier component can access a database connection pool without a proper role in its context. These types of security-centric Unit Tests are treated as *Unit Hacks* in our process and they are described later.

What parts of the Unit Test concept can be leveraged by the security team? As with other collaboration in development processes, one of the security team's chief roles is to question the development team's assumptions with regard to threats and the value of security mechanisms. Unit Tests are simple to write, and they can provide simple but powerful validation of assumptions with regard to security. Unit Test should ensure that both policy and mechanism are implemented correctly at the unit level. The granular nature of the Unit Test yields a valuable entry point for security testing beyond what is offered by system testing solely.

In the Secure Development Process Unit Tests are created out of Use Cases, each Use Case containing Pre and Post conditions. From a security standpoint, Unit Tests should be written not just to validate that the application does what it is supposed to do from a functional standpoint, but also that it performs pre condition checking such as authentication and authorization, and "ends" correctly, i.e. cleans up resources left after execution.

For more information on Unit Testing tools, refer to the Resources section.

**Code Development**

By the time code development begins in earnest the development team has a good perspective on the relevant threats the application will face, what an attack looks like and possible outcomes of an attack. Security teams are not typically engaged directly in coding business functionality, but the security team should still be involved in the development process throughout the coding. The security team should be up to speed on this phase to refine Threat Models and other artifacts, and to understand if/when key assumptions made earlier in the development life cycle are rendered invalid. During code development key architectural and business assumptions may prove to be too costly, too complex, or lose value over time. These changes in assumptions can have cascading effects towards the models developed previously so it pays for the security team to be involved to the extent that they are aware of any major changes.

**Unit Hacking Test Cases**

*Unit Hacking* applies a "white" hat approach to unit testing code. A Unit Hack Test Case is a security-centric Unit Test that executes specific security test cases designed to prove how resilient code units are to certain attacks. Unit Hack uses the same framework as Unit Tests, e.g. JUNit, Nunit, et. al., but the code is not designed to check functional behaviour, rather the Unit Hack validates security properties.

A Unit Hack should follow the same format as a Unit Test, but check for security events and properties. As with other security-centric artifacts, Unit Hacks can be derived from Misuse Cases and Threat Models, but there are many possibilities for sources of Unit Hacks; for example, when developing web based applications consider utilizing OWASP's vulnerability list for web-based applications [2] as a starting point.

Why are Unit Hacks a special case and not treated as "normal" Unit Tests? While many Unit Tests ordinarily perform basic security checks, e.g. whether or not a user can perform certain functions based on role it is still valuable to break out Unit Hacks as a special category worthy of consideration for two main reasons:

- It is a different mind set to write a Unit Hack that probes an application's defences for weaknesses, rather than a Unit Test that tests business logic's veracity.
- The operating environment and timing for a Unit Hack may need to be partitioned off from standard Unit Testing. This is due to the fact that if a Unit Hack "succeeds" then the system could be in an unusable state, user accounts could be locked, data could be corrupted. While finding this out early in the development life cycle is one of the chief benefits of the Unit Hacking approach, it is not meant to drag development to its knees. So the Unit Hacks that have a likely capability to hinder development should be executed with caution and not necessarily in conjunction with normal Unit Tests, which are frequently run at every build.

From a security standpoint Unit Hacks provide similar value as Unit Tests do to the development team. They are an early indicator of code quality, and they show a picture of the resiliency of the application from a bottom up perspective. From a pragmatic viewpoint the security team can achieve better reusability by establishing generic frameworks of Unit Hacks that the development teams can implement for patterns of applications. In other words, the security team can define a suite of Unit Hacks for web-based applications, another for asynchronous messaging, and another for rich client applications.

The following example (see Figure 1) uses JUnit to show a simple Unit Hack Test Case that checks for the validation of a password length rule. The Test Case tries a short password and expects to return false, then a tries a longer password which meets the minimum length rule and returns true. Note that since the assertions expect to return false for the *badInput* case Junit will not report these tests as errors.

**Countermeasure Development**

During code development, the security team should collaborate directly with development on countermeasures. The countermeasures are defined in the artifacts from earlier phases, such as Use Cases, Misuse Cases, and Class diagrams. Due to the iterative nature of modern development methodologies the security team should stay particularly plugged into the development and implementation of security mechanisms to ensure that the design's integrity holds. Since countermeasures are often a combination of code and configuration, the security team must oversee efforts on both sides and be in a position to adjust and refine design elements when necessary. Countermeasures are integrated into the code and configuration during the coding phase and each countermeasure implementation should be tracked as part of the configuration management process to determine its effect on the system stability, testing, and ability to defend against security test cases.

**Detection & Signature Development**

Another benefit of having security involved throughout the development life cycle is that the security team can take advantage of an inside out view of the application and can to a real degree "program" certain responses to facilitate better security once the application is operational. A good example of this is that the security team through Unit Testing and Unit Hacking builds a good understanding of how the application responds during normal operations, and exceptional conditions such as security exceptions. The security team is now in a position to analyse exceptions and understand the application, not just its requirements and its configurations, but its "living, breathing" responses to normal usage as well as malfeasance. If the security team has the requisite infrastructure it can be beneficial to deploy and test the application to build a database of responses to test conditions. Assuming a test and user base similar to production this strategy can pay off when debugging production problems. Attack signatures for specific types of attacks can be derived by building Unit Hack Test Cases that execute known attack types from the wild; and then capturing the application's responses.

Additionally, signature development for certain types of attacks that cannot be totally remediated for technical, business and/or political reasons can be a huge win for a security team to get insight into detecting security events. The implementation of the signature will vary depending on implementation, but even a humble log message that is written to identify a certain type of scan or attack can make a large improvement in the quality of detection.

**System Testing**

Beyond Unit Testing, system testing tests the quality of the application and all its component parts working together as a whole. The security concerns addressed in artifacts such as the Threat Models and Misuse Cases should also be represented in System Testing to ensure that the countermeasures and detection mechanisms function as appropriate. The system test enables the team to see how well the application performs in a variety of conditions and can also illuminate emergent properties due to integrating the platform. As with other parts of the development life cycle, the security team should be prepared to participate and iterate as necessary.

Another standard part of the system test is typically a penetration test that scans for and attempts to exploit vulnerabilities. The same types of attacks that are implemented as Unit Hacks should be attempted at system test time to ensure that the combination of components does not weaken security.

## Deployment Phase

Many times application security does not begin until the application is beginning its transition to production. The security team is called in by the development team and asked to review the application before it goes live. In our hypothetical Secure Development Process we can make use of a

```
import junit.framework.TestCase;
public class LoginTest extends TestCase {
   public void testPasswordRule {
        LoginForm lForm = new LoginForm();
        boolean badInput = lForm.validatePasswordLength("password");
        boolean goodInput = lForm.validatePasswordLength("AL0ngPa$$Word");
        assertTrue(badInput == false);
        assertTrue(goodInput == true);
        ...
   }
...
}
```

*Figure 1* – **Unit Hack Test Case**

number of artifacts to have a more precise reading of how secure the application is by this point in the development life cycle. So the deployment phase should more consist of finalizing artifacts, continued iterations, and preparing for incident response and other operational issues.

**Build and Configuration**
There are many viewpoints to consider in securing a complex system; the security team must address the various types of application users and administrators. An additional special case for consideration is the application build and configuration roles. As the application moves toward production readiness the security team should codify secure build and configuration processes and procedures. Particularly important in this respect is the ability to monitor and trace build and configuration creation and change activities. Where the technology supports it, code signing and secure classloaders can be an advantage in locking down application build. Similar to application architecture, the trust relationships involved in the build and configuration process should be examined and designed with security in mind. From a process design standpoint, the separation of privileges applies to not allow developer access directly to operational environments.

**Operational Planning and Readiness**
Operational planning for application deployment includes a variety of tasks from writing user and administrator manuals, to performing user training, finalizing documentation, and configuring administrator and monitoring tools. Similar to the build and configuration process design considerations, the security team should now partner with the operations team to understand where to integrate their processes and where separate privileges are required.

**Security Baseline**
The minimum security baseline provides a basis for secure operations as well as for auditing systems. There are a number of security baselines available from vendors as well as independent organizations like SANS. The Security baseline should reflect the design goals of the development, architecture and security teams. The security baseline artifact guides future maintenance and development activities. As with other artifacts, the security baseline should be considered a living artifact and consistently refined and updated over time.

**Response Planning**
Security is not all prevention and detection. The response plan is critical as well. While this article focuses on security in the development life cycle, it is critical that prior to deployment response planning is considered. Through participating in testing activities, and understanding key attack signatures, the security team has gained a more exact representation of some of the important events it will need to respond to. As with other operational plans the security response plans should include chain of command, for security incident response planning evidence gathering mechanisms and evidence handling are particularly critical.

**Integration into the Enterprise Security Architecture**
The application needs to be integrated and validated to work with Enterprise Security Architecture elements. These can include identity management, firewall and IDS systems. As part of systems testing each integration and any security considerations (as represented in artifacts) should have one or more corresponding test case(s) and activities.

## Conclusion
Our hypothetical Secure Development Process presented a number of areas in standard development processes which security teams can further leverage. In addition we looked at ways to extend artifacts and activities in a more security-centric way so that security can be a proactive partner in the development life cycle.

**Finding the Balance**
How much security is enough in a given piece of software? What is the right amount of involvement of security in the development life cycle? How should my organization balance functionality, risk, cost and speed? While no process has the answers to these questions today, collaboration in development between security and development, security taking the initiative to create and validate valuable artifacts can lead to a better balanced risk equation in the enterprise.

**Resources**

*Unit Testing Tools*
JUnit: Unit test tool for Java
http://www.junit.org

NUnit: Unit test tool for .Net languages
http://www.nunit.org

PyUnit: Unit test tool for Python
http://pyunit.sourceforge.net

*References*
[1] Junit Test Infected: Programmers Love Writing Tests
http://junit.sourceforge.net/doc/testinfected/testing.htm
[2] OWASP Top Ten Project
http://www.owasp.org/documentation/topten.html

## About the Author
Gunnar Peterson is CTO of Arctec Group (http://www.arctecgroup.net) a focussed, best-in-class IT consulting provider whose primary service is delivering pragmatic, objective, vendor-independent management and architectural consulting services for business-critical systems.