

Integrating security into a development process

BlackHat Windows 2004

Introduction

- _ my name is Matt Hargett
- _ co-founded BugScan, Inc. a year ago
- _ 7 years of experience in the trenches
- _ large and small companies and teams
- _ monolithic, semi-agile, and just plain anarchic processes with local and remote developers

Topic Summary

- Requirements
- Design
- Development
- QA
- Deployment

Sub-topics

- _ what usually happens
- _ problems that result
- _ proposed solution

Requirements: Usually...

- _ we only get positive stories and use cases
- _ use cases specify multiple features
- _ security requirements do not surface until a consultant is brought in later in the cycle

Requirements: Problems

- positive-centric requirements usually results in positive-centric design, coding, and testing
- negative requirements later in the cycle result in attempts to duct tape security onto the side
 - resulting in poor integration
 - poor quality and testing
 - schedule slippage

Requirements: Solution

- from the start, develop at least one negative use case for every positive use case
- positive: User is prompted for logon information
 - logon information can only contain alphanumeric characters, error is reported to user if those characters are present.
 - logon information must be shorter than 50 characters, error is reported to user otherwise.

Design: Usually...

- we do some textbook Analysis & Design
 - Get nouns from requirements, those are our objects
 - Get verbs, those are our methods (when it makes sense)
- user is prompted for logon information, user enters logon information which is securely sent to the application.
 - User->logon(), User->logoff(), etc
- maybe some header files are written, sometimes some class diagrams are drawn, but coding basically begins immediately

Design: Problems

- convoluted class structures, which in turn means a brittle design
 - duplication
 - inappropriate intimacy
- lack of understanding where and how data flows through different objects
 - larger attack surface
 - potential performance issues which can result in DoS

Design: Solution

- _ make UML class diagrams
 - easy to see various anti-patterns without any code
- _ make UML sequence diagrams for common cases
 - how many contexts does data pass through?
 - how many times does remote data marshalling occur?
 - is sensitive data encrypted during the trip?
- _ CASE tools aren't necessary to model
 - agile modeling
- _ prove the design with a little code
- _ we now have a holistic view for a secure design

Development: Usually...

- _ we code in the boundaries of our design
- _ ... until we think we're done
- _ then we fix bugs as reported

Development: Problems

- _ UML makes sure you're coloring inside the lines, not that the right crayons are used
- _ we are scared to fix problems because we "might break something"
- _ lack of objectivity for "done"

Development: Solution

— Test Driven Development

- ensures clean, testable, extensible design
- get functional regression testing for free

— Customer advocate/project manager defines "done"

— A true story of SQL Injection

- example 1

— Use Mock Objects for exception testing

- example 2

Development: Example 1 Code

```
public void SQLFilter (string str)
{
    str.Replace("'", "_");
    str.Replace(";", "_");
    str.Replace("%", "_");
}
```

Development: Example 1 Test

```
[Test]
public void testSQLFilter ()
{
    string str = “;%’”;
    SQLFilter(str);
    Assertion.AssertEquals(
        “Not all chars filtered”,
        str,
        “_____”);
}
```

Development: Example 1 Demo

The screenshot shows the NUnit GUI for the project 'BugScanWebv1Tests.dll'. The test tree on the left shows a failure in 'SQLFilterTests.testSQLFilter'. The 'Errors and Failures' pane displays the following error message:

```
BugScanWebv1Tests.SQLFilterTests.testSQLFilter :  
String lengths are both 3.  
Strings differ at index 1.  
  
expected:<__>  
but was:<'_>  
      ^
```

The status bar at the bottom indicates: Completed, Test Cases : 35, Tests Run : 35, Failures : 1, Time : 2.7439456.

Development: Example 2 Code

```
[Test]
public void testDataBaseException()
{
    MockControl control;
    DB mockDB;
    User user;

    control = EasyMock.ControlFor(typeof(DB));
    mockDB = (DB)control.GetMock();
    mockDB.Auth("user", "pass");
    mockDB.SetVoidCallable();
    mockDB.ChangeDB();
    mockDB.SetThrowable(new SystemException());
    control.Activate();

    user = new User(mockDB);
    user.Logon("user", "pass");
    Assertion.AssertEquals(
        "database error shouldn't yield authenticated user",
        false,
        user.IsAuthenticated());
}
```

Development: Solution (cont'd)

- _ TDD gives us the agility to deal with security bugs in a timely fashion
- _ helps us focus on independent objects and well-defined interfaces
- _ which in turn allows us to do negative testing in fast, automated way in the core logic before a UI even exists to pen-test

QA: Usually...

- _ creates a large test plan document
- _ works "stupid hard"
- _ has responsibility without authority

QA: Problems

- _ duplicates use case artifacts that already exist in large, unmanageable documents
- _ can't really measure where they are
- _ doesn't have the time or knowledge to set up complex environments
- _ functional testing gets held up by instability
- _ burn out and hopelessness

QA: Solutions

- use a common store for use case artifacts, shared between business and engineering
- QA should create positive and negative use case variants from the beginning
- most long hours are repetitive manual testing, invest time in automation up front
- create a smoke test code must pass to be tested
 - minimum code coverage by unit tests (PureCoverage)
 - no unit test runtime bugs detected (Purify, Insure++)
 - static analysis (PC-Lint, BugScan, etc)
 - integrate smoke test into automated build
- this gives QA time to focus on more complex and negative scenarios

Deployment: Usually...

- _ install it or put it up for download and forget about it
- _ sometimes blackbox fault injection and/or code review is done
- _ we choose one module to focus on since we don't have enough resources

Deployment: Problems

- _ deep knowledge can be required for fault injection to produce any results
- _ we can't get source code to review due to political problems
- _ we can't push our tools or process further into the development groups

Deployment: Solutions

- _ Do static analysis for security problems first
 - helps direct manual reviews
- _ Then focus on runtime analysis
 - fault injection and code coverage
- _ Use binary analysis tools
 - in conjunction with source analysis
- _ CSO/CTO is given “stop ship” authority

ChangeLog

- Negative requirements
- A little additional modeling
- Unit testing via TDD and Mock Objects
- Reuse of existing use case artifacts in QA
- Use of static and runtime analysis
- Give the right people the authority to do the right thing

Bibliography

— Microsoft Solutions Framework

- www.microsoft.com/msf
- www.learnvisualstudio.net

— Extreme Programming

- www.xprogramming.com

— Agile modeling

- www.agilemodeling.com

— Test-Driven Development

- [TDD: A Practical Guide](#)

— NUnit

- www.nunit.org

— EasyMock.NET

- www.easymock.net

— BugScan

- www.hbgary.com

— PureCoverage

- www.rational.com

Contact

— Email:

- matt@hbgary.com

— Blog:

- www.rootkit.com