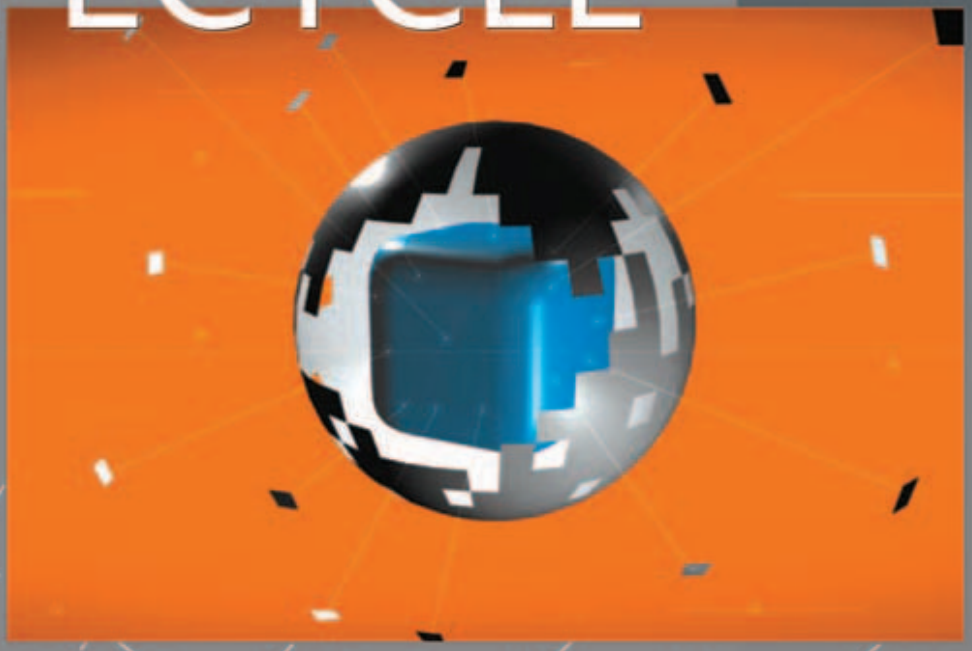
Microsoft

secure software DEVELOPMENT SERIES

BEST PRACTICES

THE SECURITY DEVELOPMENT LIFECYCLE



SDL: A Process for Developing Demonstrably More Secure Software

Michael Howard and Steve Lipner

Foreword by Jim Allchin

Co-President, Platforms & Services Division, Microsoft Corporation

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number 2006924466
978-07356-2214-2
0-7356-2214-0

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWE 1 0 9 8 7 6

Distributed in Canada by H.B. Fenn and Company Ltd.A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Devon Musgrave
Technical Editor: Virgil Gligor; Technical Review services provided by
Content Master, a member of CM Group, Ltd
Copy Editors: Bill Bowers & Shannon Leavitt
Indexer: Richard Shrout

Body Part No. X11-74982

Table of Contents

Chapter 18

Integrating SDL with Agile Methods

In this chapter:

Like them or not, Agile methods and processes such as Extreme Programming (XP) and Agile processes such as Scrum are gaining popularity (Extreme Programming 2006, Schwaber 2004). Microsoft has also adapted its Microsoft Solutions Framework to include Agile methods (Microsoft 2006).

We're not going to debate the merits of these rapid-development processes, but groups within Microsoft, such as those in MSN and Windows Live, have integrated Agile methods into their development processes to good benefit. What sets the MSN and Windows Live projects apart from most Microsoft projects is that MSN projects are not huge development efforts such as Microsoft Windows or Microsoft Office. Complex to a degree, they have an important goal: rapidly developed small releases. Examples of projects delivered by MSN using Agile methods include

- MSN Messenger 7.5
- MSN Tabbed Browsing for Microsoft Internet Explorer
- MSN Anti-Phishing add-in
- MSN Support tools
- Internet Access RADIUS Service

Note that some of these products were built using only Agile methods and others experimented with various ideas from Agile methods.

The rest of this chapter is split in two parts, the first looking at Security Development Lifecycle (SDL) concepts and applying them to Agile methods, and the second looking at Agile methods with regard to adding SDL concepts. Please note that the goal of this chapter is not to cover every aspect of all Agile methods. Rather, it is to choose where it makes sense to augment the rules and practices of Agile methods with more security discipline and best practices.

Using SDL Practices with Agile Methods

In this first section, we'll look at the core SDL practices and consider how these can be used with Agile methods.

Security Education

Regardless of what software development method you employ, security education is critical. No development method will create secure software if the people building the software do not use simple security best practices. We've heard people claim that <*insert popular development method*> produces bug-free software. This might be true—and of course, it is true if you know nothing about security bugs, because you wouldn't recognize a security bug if you had no idea what one was.

Hence, you should follow the standard SDL policy and train all engineers about security issues at least once a year. In the overall cost of software development, the cost of education (in terms of time and effort) is tiny, and the risk of security errors being introduced is large.

Tip We appreciate that everyone developing software is in a hurry these days, but please do not skimp on security and privacy education.

Because of the less structured environment fostered by Agile development, the MSN teams push for more time spent on education and training. As a result, one of the MSN group's new requirements is that at least one hour be spent every two weeks on training and education. Of course, security is not the only possible subject that could be covered, but it is an important component.

Important We would argue that security education is more critical in the Agile environment because more decision-making power is placed in the hands of the product owner and development team.

One could justifiably argue that the XP concept of pair programming would aid with security education. But if neither member of a pair understands security, chances are that neither will notice a security bug. It is our opinion that all engineers should have classroom-style or online security education. It really is that important.

Project Inception

Contrary to popular belief, Agile methods do require some up-front groundwork. From an SDL perspective, the team must understand who the security go-to person is. This person is the *security coach*.

Note The SDL concept of "security advisor" translates nicely to an Agile "security coach."

Another part of XP is the notion of moving people around. If you adhere to this principle, consider moving the security coach around so you will force more people to take a security leadership position. However, do not take unnecessary risks in choosing the security person: this person has to make the best-possible security decisions for the product.

Establishing and Following Design Best Practices

Design, according to the traditional software-engineering definition, does not exist in most Agile methods. Rather, as the application develops or is iterated, the design is also iterated. Of course, you could always make serious design mistakes early in the product's life, but the goal of Agile development is to understand these mistakes early, in conjunction with customers, and make incremental changes for the next iteration. Often an iteration, or *sprint* (in Scrum parlance), might be only 14 or 30 days long.

Another aspect of many Agile methods, including Extreme Programming, is *simple design*. The software should include only the code that is necessary to achieve the desired results, as communicated by the customer. Simple design has a valuable security side effect: if you keep the design simple, you increase the chance that the design is secure. Complex software is difficult, if not impossible, to make totally secure. Also, smaller and more modular software is likely to be architecturally more secure.

The core of the Agile design philosophy is the *user story*. A user story is a short text that describes how the system is supposed to solve a problem or support a business process. User stories should encompass the customer's security concerns. Developers sign up for stories, and it's not unreasonable to expect one or more stories to focus solely on the security of the system. But a story about security should focus on threats perceived by the customer, which we will discuss next.

Best Practices For some development projects, procuring an on-site customer might be impossible. Very large projects, such as development of an operating system or a Web server, are examples. In cases like these, consider using personas, which you create based on real customer data, to help prioritize features and maintain focus on target customers (Kothari 2004). Above all, personas must be believable! You can also dedicate an employee to play the role of each of the assigned personas in person during meetings.

Risk Analysis

When building an application using Agile methods, you will probably not have a data flow diagram (DFD). In some software projects, there is a design sprint, and a deliverable from the design sprint could be a DFD.

But at some point, you will know that component A will communicate with component B using, say, sockets, and that component B uses a database to persist the data over, say, Open Database Connectivity (ODBC). Figure 18-1 shows an example of this arrangement.

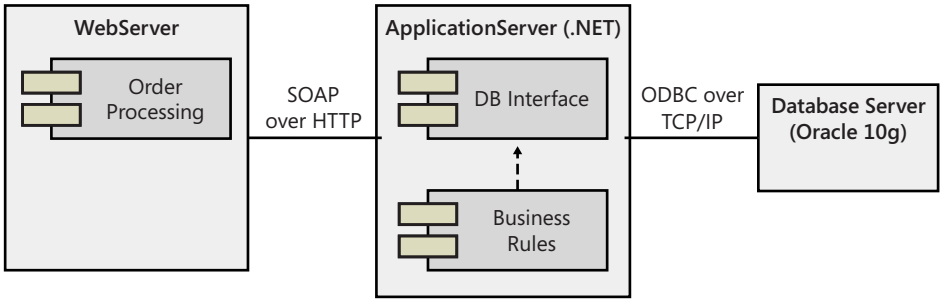


Figure 18-1 A portion of a story showing interaction among various components.

With this small diagram in hand, you can easily apply the risk analysis process using the following mapping:

- Code portions of the diagram are processes.
- Users are external entities.
- Any place where data is persisted is a data store.
- Interaction between code or data stores is a data flow.
- Interaction between users or external entities and code is a data flow.

Now you can apply the STRIDE threat taxonomy versus DFD elements described in Chapter 9, "Stage 4: Risk Analysis," and ask the customer questions such as the following:

- Does it concern you that an authenticated user or attacker can read any data from the Sales Order database?
- Will you be concerned if a valid user is denied access or degraded in her use of the application server?
- Does it concern you that anonymous users can read and change the network traffic between the application server and the database server?

If the answer to any of these questions is yes, that answer becomes part of the story. If not, make a note in the story that the customer is not concerned.

Best Practices Translation from threats in the threat model to questions to ask the customer is the job of the security coach.

Take a closer look at the question sentences:

- "Anonymous," "authenticated user," and "valid user" are examples of roles or trust levels.
- "Read" is a synonym for information disclosure (I in STRIDE). "Change" means tampering (T in STRIDE). Denied or degraded service is an example of denial of service (D in STRIDE).
- "Sales Order database" and "application server" are example processes you need to defend from attack. Always remember that a customer's machine is an asset that always requires protection.

You can apply this simple analysis method to all parts of the Object Management Group's UML (Unified Modeling Language) diagram. In short, rather than thinking of potential security issues in an ad hoc manner, this method combines the analytical threat-modeling technique with rapid Agile development methods.

Creating Security Documents, Tools, and Best Practices for Customers

Agile methods are often criticized for having very little user-oriented documentation. At the very least, you should provide important security best practices in online Help files and within the application's user interface. Better still, if you are using the risk analysis process described in Chapter 9, you can use the security notes to help derive customer-facing documentation. That being said, it all depends on whether this is what the customer wants. So ask your customers what they want. Chances are that if you have a substantial user base (such as that of MSN Messenger 7.x), you should simply do the right thing by providing security best-practice documentation because no customer actively wants users to make security mistakes.

Secure Coding and Testing Policies

Agile methods support the notions of coding practices and requiring constant testing. In the case of coding practices, you should adopt secure coding best practices defined by SDL, such as the following:

- Requiring coding best practices.
- Not using banned application programming interfaces (APIs). (See Chapter 19, "SDL Banned Function Calls.")
- Using only appropriate cryptographic algorithms. (See Chapter 20, "SDL Minimum Cryptographic Standards.")
- Using static analysis tools such as those included with Microsoft Visual Studio 2005. (See Chapter 21, "SDL Required Tools and Compiler Options.")

Better yet, don't just define and use the coding rules; if you use Microsoft Visual Studio 2005 Team System, set up check-in policies and testing policies that enforce your rules (Microsoft 2005a, Microsoft 2005b).

Testing is a little more involved. Extreme Programming mandates that if you find a bug, you should write a test; this mandate applies to security bugs also. For example, if you find an integer overflow such as the following in your C/C++ code, you must build a security test that triggers this bug.

```
void * RenderEngine::AllocArbitraryBlob(int qty, int size) {
   if (qty && size)
      return GlobalAlloc(0,qty * size);
   else
      return NULL;
}
```

You must fix the code and rerun the test. The test should not fail. Rerun the test on every new build of your code. In CppUnit-like pseudocode (Wikipedia 2006, CppUnit 2006), your test might look like the following code example:

```
// Instantiate the class under test.
RenderEngine *e = new RenderEngine();

// Zero quantity or size is a no-op.
CPPUNIT_ASSERT(e->AllocArbitraryBlob(0,10) == NULL);
CPPUNIT_ASSERT(e->AllocArbitraryBlob(10,0) == NULL);

// An overflow should fail with NULL.
CPPUNIT_ASSERT(e->AllocArbitraryBlob(0x1fffffff,0x10) == NULL);

// A signed versus unsigned overflow should fail with NULL.
CPPUNIT_ASSERT(e->AllocArbitraryBlob(0x1fffffff,-1) == NULL);
CPPUNIT_ASSERT(e->AllocArbitraryBlob(-1,0x1fffffff) == NULL);

// This should succeed; NULL means there was an int overflow.
CPPUNIT_ASSERT(e->AllocArbitraryBlob(0x1fffff,1) != NULL);

// This should succeed too.
// And we need to verify that the return buffer size is correct.
void *ptr = e->AllocArbitraryBlob(0x200,0x20);
CPPUNIT_ASSERT(0x200*0x20 <= GlobalSize(ptr));
GlobalFree(ptr);
```

Then you would make the code fix:

```
inline void * RenderEngine::AllocArbitraryBlob(size_t qty, size_t size) {
    size_t alloc = qty * size;

    if (alloc ==0)
        return NULL;

    // Function is inlined, so 'size' is typically a constant
```

```
    // and the division is optimized away at compile-time
    if (MAX_INT / size <= qty)
        return GlobalAlloc(GPTR,alloc);
    else
        return NULL;
}
```

When you rerun the tests, they should all succeed with the defensive code in place. You should build tests like this for all bugs, including security bugs.

Finally, fuzz testing lends itself well to Agile methods. If you have code that parses any input, you should build fuzz tests for all the entry points. These should be run daily, just like every other test.

Security Push

Within most Agile methods, there is no concept of specialized coding events such as those focusing on usability or security. However, a critical tenet of Extreme Programming is refactoring, which concerns itself with improving the internal representation of the code to make it cleaner, easier to read and maintain, higher quality, and, in our opinion, more secure (Fowler 2005). Secure software is by definition quality software, after all. One could argue there is no need for security pushes when Agile methods are used, except in one particular case: the security push, as defined in the SDL, focuses almost exclusively on legacy code. Code that has not been touched in three or more years probably has security bugs because

- The security landscape evolves substantially for good and for ill, but mostly for ill.
- Security tools advance quickly for good and for ill.
- People generally get better at finding security bugs, for good and for ill.

If the legacy code handles sensitive or personally identifiable data or is exposed to the Internet, *all* the legacy code should be reviewed in a series of "refactoring spikes" until all the code is reanalyzed, new tests are built, and bugs are fixed. More information about refactoring is provided later in this chapter.

If you use Scrum, you should also consider adding legacy code cleanup work to the product backlog every couple of sprints. The product backlog is a list of all the desired changes to the product being developed. Work items are taken from the product backlog and added to the sprint backlog by the product owner. If this is the first time your product has been subjected to security rigor, you should make the previous code cleanup work a major component of the backlog.

The MSN team has a mini-security push prior to a Release Candidate in which there is a group security code review and a dedicated test cycle for security testing. This amounts to one day for a two-week sprint or two days for a month-long sprint.

Tip Some proponents of Agile methods at Microsoft indicate that having a series of one-day "security days" in the middle of the development schedule is beneficial.

Final Security Review

The Final Security Review (FSR), as discussed in Chapter 14, "Stage 9: The Final Security Review," is the point at which you verify the product is ready to ship from a security and privacy standpoint. Agile methods cannot employ a full-fledged FSR because of time constraints, but it does not mean you cannot do an FSR! For code developed using Agile methods, we propose the following minimum set of FSR requirements:

- All developers working on this iteration have attended security training within the last year.
- Unfixed security-related bugs are in fact appropriate to leave in this release. If the customer is well defined, the customer should have the final say.
- All customer security stories have been implemented correctly and signed off by the customer.
- All secure-coding best practices have been adhered to.
- All code-scanning tools have been used, and appropriate bugs have been fixed.
- All security-related tests have been run and bugs fixed.
- All parsed data formats have fuzz tests.
- If you are using managed code, such as C# or Microsoft Visual Basic .NET, results from tools like FxCop are evaluated and, if need be, fixed.
- Compilers used meet the minimum SDL requirements. (See Chapter 21.)
- If you are using Visual Studio, all C/C++ code is compiled with */GS* and linked with */SafeSEH*.

It's important that all security-related user stories be evaluated to make sure they are implemented correctly and meet the customer's needs.

All of the items in this list should be on a Big Visible Chart (BVC), also called an Information Radiator (Jeffries 2004). An important part of Extreme Programming is communication, and BVCs are a good way to very openly communicate what is expected of the engineering team.

Finally, because of the highly iterative nature of Agile methods, you can break an FSR into small "feature FSRs." In other words, rather than putting the entire software product through the FSR process every time you iterate, perform smaller FSRs on one or two features every sprint until the entire product is reviewed. The review order is determined by risk, and the riskiest features are reviewed first.

Product Release

An important part of the scheduling process when you use Extreme Programming is the release plan. This plan should include which security-related stories must be delivered to customers before you can consider the current iteration complete. When all these stories are complete, the product is ready for release to the customer.

Security Response Execution

The Security Response Execution stage is unique to SDL and is not apparent in Agile methods. Agile methods support the concept of rapid iterations that have well-defined and customer-supported features and the notion that any bugs found in one iteration can be fixed in the next iteration. But here is the problem: security bugs are not typical bugs. They might very well lead to emergencies that can put the customer at risk, which means you need to have a plan in place to handle potential security bugs at once. The preferred way to treat this situation is as a spike. You use a spike solution when you are working in a new problem domain or with a new technology you do not understand. We would argue that newly discovered security bugs fit both of these conditions. They are new problems in that the instance of this bug is new to you and your customer, and it's something you might not yet understand how to fix correctly. Another reason to use a spike is time; remember, if a security bug is publicly known, the chance that the vulnerability could be used to attack your customer increases over time until the customer applies the fix, mitigation, or workaround. Therefore, we recommend that the spike have two major components:

1. A viable workaround as soon as possible.
2. A real code-level or architecture-level remedy.

As a first step, determining an appropriate workaround might include tasks like these:

- Enabling a firewall rule
- Turning off some functionality
- Employing another security feature

When creating the real remedy, which might be a design or code change, it's important that you create a test to detect the defect first. Then make the fix and rerun the test to verify that the fix works.

Here is where Extreme Programming and SDL might be perceived to diverge. A spike is supposed to be a very discrete event focusing on solving one technical problem, but in the case of a security defect, the chances are good that the same type of bug exists in more than one place in the code. Because of the way security researchers find security bugs, they *will* find the other bugs—guaranteed! So when you find a security bug, you should form a spike that includes a security expert, make the appropriate and correct code fix (and the test), and then find the other defect variants within the same code area. Don't forget to create small tests of all the bugs.

Once the fix is complete and deemed acceptable, you must issue a fix and provide guidance to your customers.

Core values of Agile methods include learning from mistakes and being adaptive rather than predictive. These notions apply to security bugs, too; you must apply a root-cause analysis to answer the following questions:

- Why did this mistake occur?
- What do we need to change to make sure this mistake never happens again? The answers to this might include better testing, more education, and changes to and enforcement of the best practices.
- Can a tool be created to search for the mistake in future code?
- Where else could this mistake have occurred?

You should apply your new knowledge to all future iterations to reduce the chance that the same mistake is made again (and again!).

Augmenting Agile Methods with SDL Practices

In this short section, we'll look at some of the Agile doctrines and see how they can be augmented with security best practices from SDL. The following list identifies the Agile doctrines that we'll look at:

- Planning
    1. User stories
    2. Release planning
    3. Small releases and iterations
    4. Moving people around
- Design
    1. Simplicity
    2. Spike solutions
    3. Refactoring
- Coding
    1. Constant customer availability
    2. Coding to standards
    3. Coding the unit test first
    4. Pair programming

5. Integrating often
6. Leaving optimization until last

- Testing all bugs

Let's look at the specific doctrines in detail.

User Stories

User stories should include the customer's security requirements. As previously noted, such stories must be based not on intuition but on real-world threats. Use the risk- and threat-modeling method outlined in the "Risk Analysis" section in this chapter to understand these threats and articulate them to customers.

In his book *User Stories Applied: For Agile Software Development*, Mike Cohn suggests adding "Constraints" to user stories (Cohn 2004). A constraint is something that must be obeyed and is fundamental to the business. For example, from a security perspective, a story might include directives such as these:

- "The software must not divulge the data in the Orders database to unauthorized users."
- "All software add-ins must have valid digital signatures in order to run within the system."
- "The client must always authenticate the validity of the server."

For a software product to be complete, all user stories should be complete. By "complete" we mean

- All code and test code for each story is checked in.
- All unit tests for each story are written and passed.
- All applicable functional tests for each story are identified, written, and passed.
- Product owner has signed off.

And, from an engineering practices perspective, "complete" means the following steps have been taken:

- All appropriate security best practice has been adhered to, or exceptions granted.
- The latest compiler versions are used.
- All code scanning tools have been run over all code.
- All bugs from the code scanning tools are fixed or postponed.
- There is no use of banned functionality.

Small Releases and Iterations

It is easier to secure a small code delta than a large code delta. It is common to see coding bugs of all types on the boundary of old and new code; if this boundary is kept small, bugs can be found relatively easily. The doctrine of small releases is good for security, too. Another benefit of small iterations is that you can prioritize security defenses. Critical defenses can be added to the code in the current iteration, and less-important defenses can be added to later iterations if needed. Small iterations also address the notion of not adding functionality earlier than it's needed.

We have learned the hard way that one drawback of introducing a new security defense is that the chance of also introducing functional regressions is very high. Be forewarned.

Moving People Around

In general, competent security specialists are scarce and hard to hire. Be prepared to wait to hire the right person. Once you have hired an effective security person, encourage him to teach security to others in the team. A critical component of security skills is education: have the guru teach and mentor others in the team.

Note that although moving people around is a good idea, the authors have yet to see any team do it.

Best Practices Security should be a skill common to all software developers, not confined solely to just a select group of specialists.

Simplicity

A simple application is more secure than a complex application, period. Complexity is an enemy of security. Of course, in the real world, this truism is a little more subtle. We can always write simple software that would never get the job done. In fact, most code today is complex because business processes are complex and have thorny, but necessary, requirements that add complexity to the code, such as responsiveness, timeliness, robustness, transaction processing, offline and online capabilities, integration with older systems, and so on. But at the micro-level, your code can be simple and easy to understand and, hence, to maintain. Where possible, strive for simple designs and easy-to-understand code.

Spike Solutions

Invariably, you'll hit security roadblocks, perhaps security bugs or your own uncertainty on the best way to implement or take advantage of a security feature. A spike solution is a great method to determine the best way to resolve security dilemmas. Take two developers off the core project to work on the security solution.

Refactoring

At Microsoft, we often systematically review older code, looking for security bugs; if issues are found, the code is fixed. In some cases, design issues or erroneous coding patterns are found, and these patterns are fixed. This concept is very similar to that of *refactoring*, which is a technique for restructuring or changing an existing body of code without changing its interface or external behavior. You must consider security bugs as part of your refactoring process. Examples of security refactoring include

- Replacing banned APIs with safer APIs; for example, replacing strcpy with StringCchCopy or strcpy_s. (See Chapter 19.)
- Replacing weak crypto algorithms with more up-to-date and secure versions. (See Chapter 20.)
- Making cryptographic code more agile by removing hard-coded algorithm names, key sizes, and other cryptographic-related settings. (See Chapter 20.)
- Replacing integer arithmetic used in memory allocations and array indexing with safer code.

There are challenges with refactoring for the sake of refactoring—most notably, defects, usually called regressions, could be entered into the code base (Garrido and Johnson 2002).

Constant Customer Availability

The customer is a key contributor (some say the only contributor) to user stories. The customer must also provide the security requirements for the stories. You can make sure nothing is missing from user stories by building threat models for components within the application and validating that no threats are missing from the customer's stories. However, to many customers, security is an unspoken requirement. You really have to probe customers to learn how much security they'd like to buy. Customers won't mention it—they'll just say "Make it secure!" (which, of course, is meaningless).

Important It's imperative that you always consider how the software can be misused.

When security issues arise, the customer must be consulted once the threats are thoroughly understood. At the meeting to review the threats, use a spike to determine the appropriate remedy.

Coding to Standards

Secure coding standards must be adhered to, and source-code analysis tools must be used regularly to help catch various security bugs. Refer to Chapter 11, "Stage 6: Secure Coding Policies," for secure coding ideas. The beauty of coding to standards is that you can reduce (not

eliminate) the chance that new bugs, including security bugs, are entered into the system in the first place.

Important Development and test tools for security play an important role in an Agile environment due to the absence of specifications.

Coding the Unit Test First

The "Coding the Unit Test First" doctrine is especially true of fuzz tests; for any protocol you parse, or for any payload you read and respond to, you should build a fuzz generator for that protocol or payload. Refer to Chapter 12, "Stage 7: Secure Testing Policies," for fuzz-testing concepts. The author of this chapter (Howard) believes security can be significantly improved if unit security testing becomes part of per-function or per-module unit before the application is assembled.

Pair Programming

At Pairprogamming.com, the practice is described as follows:

> *Two programmers working side-by-side, collaborating on the same design, algorithm, code or test. One programmer, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects and also thinks strategically about the direction of the work. On demand, the two programmers can brainstorm any challenging problem. Because the two programmers periodically switch roles, they work together as equals to develop software. (Pair Programming 2006)*

Having a person observe while another codes is an effective way to detect security bugs as they are entered or, better yet, to prevent them from being entered in the first place. You can help team members develop security skills by pairing them with the security expert.

Integrating Often

Integrating programmers' small code updates often will help you find security bugs faster than waiting for large code changes.

Leaving Optimization Until Last

There can be a conflict between optimization and security. Optimization itself doesn't necessarily lead to security bugs, but in our experience, making large changes to the code late in the process always leads to errors in the system. Beware.

When a Bug Is Found, a Test Is Created

In the authors' opinion, creating a test whenever a bug is found is wise because doing so helps prevent the bug from reentering the code base (a regression). Every time you identify a security bug, create a test case to find and fix the bug. Then rerun the test on every subsequent version to make sure the bug is indeed fixed.

Summary

To date, there is very little guidance for development teams wanting to augment Agile methods, such as Scrum and Extreme Programming, with security discipline. Based on our conversations with Agile proponents, most of the SDL best practices and requirements can be easily incorporated into Agile practice. Doing so can only be beneficial for those using Agile methods.

References

(Extreme Programming 2006) "Extreme Programming: A Gentle Introduction," *http://www.extremeprogramming.org/*.

(Schwaber 2004) Schwaber, Ken. *Agile Project Management with Scrum*. Redmond, WA: Microsoft Press, 2004.

(Microsoft 2006) "MSF for Agile Software Development," *http://msdn.microsoft.com/vstudio/teamsystem/msf/msfagile/*. March, 2006.

(Kothari 2004) Kothari, Nikhil. "Applying personas," *http://www.nikhilk.net/Personas.aspx*. January 2004.

(Microsoft 2005a) Visual Studio 2005 Team Server Check-in Policy. "Walkthrough: Customizing Check-In Policies and Notes," *http://msdn2.microsoft.com/en-us/library/ms181281.aspx*. MSDN, 2005.

(Microsoft 2005b) Michaelis, Mark. "Introducing Microsoft Visual Studio 2005 Team System Web Testing," *http://msdn.microsoft.com/library/en-us/dnvs05/html/VS05TmSysWebTst.asp*. MSDN, September 2005.

(Wikipedia 2006) "XUnit," *http://en.wikipedia.org/wiki/XUnit*.

(CppUnit 2006) "CppUnit Wiki," *http://cppunit.sourceforge.net/cppunit-wiki*.

(Fowler 2005) Fowler, Martin. "Refactoring Home Page," *www.refactoring.com*.

(Jeffries 2004) Jeffries, Ron. "Big Visible Charts," *http://www.xprogramming.com/xpmag/BigVisibleCharts.htm*. October 2004.

(Cohn 2004) Cohn, Mike. *User Stories Applied: For Agile Software Development*. Reading, MA: Addison Wesley Professional Co., 2004.

(Garrido and Johnson 2002) Garrido, Alejandra, and Ralph Johnson. "Challenges of Refactoring C Programs," *https://netfiles.uiuc.edu/garrido/www/papers/refactoringC.pdf*. May 2002.

(Pair Programming 2006) Williams, Laurie. "What is pair programming?" *http://www.pairprogramming.com/*.