Microsoft

Microsoft®
Security Development Lifecycle

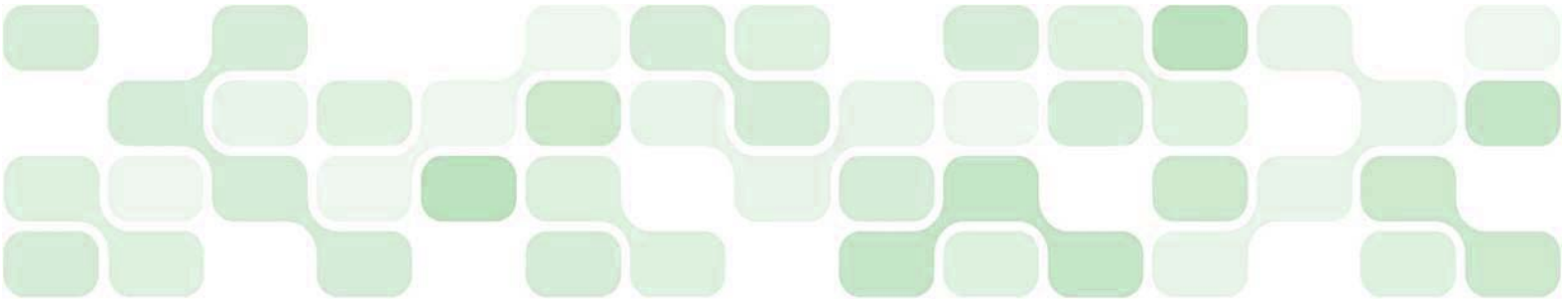
# Security Development Lifecycle for Agile Development

## Version 1.0

June 30, 2009

For the latest information, please see http://www.microsoft.com/sdl.





## Abstract

This document defines a way to embrace lightweight software security practices when using Agile software development methods, such as Extreme Programming (XP) and Scrum. The goal is to meld the proven Microsoft Security Development Lifecycle (SDL) with Agile methodologies in a way that maintains the principles of both the Agile methods and the SDL process.

This document does not explain all the nuances of the SDL. To gain a deeper understanding of the SDL, you can review the latest version at http://www.microsoft.com/sdl.

The intended audience for this document is development teams who want to build more secure applications using Agile methods. No extensive SDL or Agile knowledge is assumed.

## Contents

## Introduction

Many software development organizations, including many product and online services groups within Microsoft, use Agile software development and management methods to build their applications. Historically, security has not been given the attention it needs when developing software with Agile methods. Since Agile methods focus on rapidly creating features that satisfy customers' direct needs, and security *is* a customer need, it's important that it not be overlooked. In today's highly interconnected world, where there are strong regulatory and privacy requirements to protect private data, security must be treated as a high priority.

There is a perception today that Agile methods do not create secure code, and, on further analysis, the perception is reality. There is very little "secure Agile" expertise available in the market today. This needs to change. But the only way the perception and reality can change is by actively taking steps to integrate security requirements into Agile development methods.

Microsoft has embarked on a set of software development process improvements called the Security Development Lifecycle (SDL). The SDL has been shown to reduce the number of vulnerabilities in shipping software by more than 50 percent. However, from an Agile viewpoint, the SDL is heavyweight because it was designed primarily to help secure very large products, such as Windows® and Microsoft Office, both of which have long development cycles.

If Agile practitioners are to adopt the SDL, two changes must be made. First, SDL additions to Agile processes must be lean. This means that for each feature, the team does just enough SDL work for that feature before working on the next one. Second, the development phases (design, implementation, verification, and release) associated with the classic waterfall-style SDL do not apply to Agile and must be reorganized into a more Agile-friendly format. To this end, the SDL team at Microsoft developed and put into practice a streamlined approach that melds agile methods and security—the Security Development Lifecycle for Agile Development (SDL-Agile).

## Melding the Agile and SDL Worlds

With Agile release cycles taking as little as one week, there simply isn't enough time for teams to complete all of the SDL requirements for every release. On the other hand, there are serious security issues that the SDL is designed to address, and these issues simply can't be ignored for any release—no matter how small.

Integrating the two worlds is not as difficult as it might seem—at its heart, the SDL defines tasks, and these tasks can be mapped into an Agile development process. One benefit of the SDL is that it is relatively artifact-free, which means there is little documentation overhead (with the notable exception of threat models, which are discussed later in this document). It is possible to create artifacts if they are needed, but this is generally not required in an Agile environment.

## SDL-Agile Requirements

A workhorse of Agile development is the *sprint*, which is a short period of time (usually 15 to 60 days) within which a set of features or stories are designed, developed, tested, and then potentially delivered to customers. The list of features to add to a product is called the *product backlog*, and prior to a sprint commencing, a list of features is selected from the product backlog and added to the *sprint backlog*. The

SDL fits this metaphor perfectly—SDL requirements are represented as tasks and added to the product and sprint backlogs. These tasks are then selected by team members to complete. You can think of the bite-sized SDL tasks added to the backlog as *non-functional stories*.

## Every-Sprint Requirements

In order to fit the weighty SDL requirements into the svelte Agile framework, SDL-Agile places each SDL requirement and recommendation into one of three categories defined by frequency of completion. The first category consists of the SDL requirements that are so essential to security that no software should ever be released without these requirements being met. This category is called the *every-sprint* category. Whether a team's sprint is two weeks or two months long, every SDL requirement in the every-sprint category must be completed in each and every sprint, or the sprint is deemed incomplete, and the software cannot be released. This includes any release of the software to an external audience, whether this is a box product release to manufacturing (RTM), online service release to Web (RTW), or alpha/beta preview release.

Some examples of every-sprint requirements include:

- Run analysis tools daily or per build (see the Tooling and Automation section later in this document).
- Threat model all new features (see Threat Modeling: The Cornerstone of the SDL).
- Ensure that each project member has completed at least one security training course in the past year (see Security Education).
- Use filtering and escaping libraries around all Web output.
- Use only strong crypto in new code (AES, RSA, and SHA-256 or better).

For a complete list of the every-sprint requirements as followed by Microsoft SDL-Agile teams, see Appendix A.

## Bucket Requirements

The second category of SDL requirement consists of tasks that must be performed on a regular basis over the lifetime of the project but that are not so critical as to be mandated for each sprint. This category is called the *bucket* category and is subdivided into three separate buckets of related tasks. Currently there are three buckets in the bucket category—verification tasks (mostly fuzzers and other analysis tools), design review tasks, and response planning tasks. Instead of completing all bucket requirements each sprint, product teams must complete only one SDL requirement from each bucket of related tasks during each sprint. The table below contains only a sampling of the tasks for each bucket. To see a complete list of all tasks for all three buckets, consult Appendix B: Bucket Requirements.

| Verification Tasks | Design Review | Response Planning |
|---|---|---|
| ActiveX® fuzzing | Conduct a privacy review | Create privacy support documents |
| Attack surface analysis | Review crypto design | Update security response contacts |
| Binary analysis (BinScope) | Assembly naming and APTCA | Update network down plan |
| File fuzz testing | User Account Control | Define/update security bug bar |

*Table 1. Example of bucket categories. For a complete list of bucket items, see Appendix B: Bucket Requirements.*

In this example, a team would be required to complete one verification requirement, one design review requirement, and one response planning requirement in every sprint (in addition to the every-sprint requirements discussed earlier). For sprint one, the team might choose to complete *ActiveX fuzzing*, *Review crypto design*, and *Update security bug bar* from the table. For sprint two, they might choose *Binary analysis*, *Conduct a privacy review*, and *Update network down plan*.

It is left to the product teams to determine which tasks from each bucket that they would like to address in any given sprint. The SDL-Agile does not mandate any type of round-robin or other task prioritization for these requirements. If your team determines that they are best served by completing file fuzzing requirements every other sprint but that SOAP fuzzing only needs to be performed every 10 sprints, that's acceptable.

However, no requirement can be completely ignored. Every requirement in the SDL has been shown to identify or prevent some form of security or privacy issue, or both. Therefore, no SDL bucket requirement can go more than six months without being completed.

## One-Time Requirements

There are some SDL requirements that need to be met when you first start a new project with SDL-Agile or when you first start using SDL-Agile with an existing project. These are generally once-per-project tasks that won't need to be repeated after they're complete. This is the final category of SDL-Agile requirements, called the *one-time requirements*.

The one-time requirements should generally be easy and quick to complete, with the exception of creating a baseline threat model, which is discussed later in this document. Even though these tasks are short, there are enough of them that it would not be feasible for a team just starting with SDL-Agile to complete all of them in one sprint, given that the team also needs to complete the every-sprint requirements and one requirement from each of the buckets.

To address this issue, the SDL-Agile allows a grace period to complete each one-time requirement. The period generally ranges from one month to one year after the start of the project, depending on the size and complexity of the requirement. For example, choosing a security advisor is considered an easy, straightforward task and has a one-month completion deadline, whereas updating your project to use the latest version of the compiler is considered a potentially long, difficult task and has a one-year completion deadline. The current list of one-time requirements and the corresponding grace periods can be found in Appendix C of this document. Figure 1 provides an illustration of this process in action.

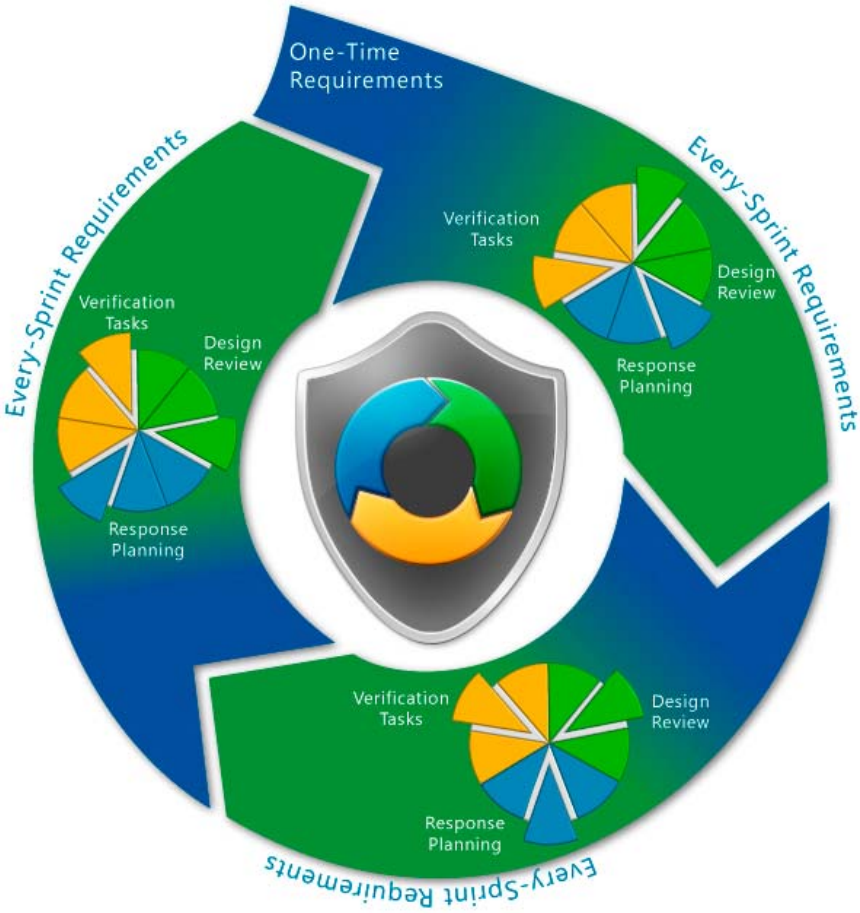



*Figure 1. SDL-Agile process*

## Constraints

The main difficulty that SDL-Agile attempts to address is that of fitting the entire SDL into a short release cycle. It is entirely reasonable to mandate that every SDL requirement be completed over the course of a two- or three-year-long release cycle. It is not reasonable to mandate the same for a two- or three-*week*-long release cycle. The categorization of SDL requirements into every-sprint, one-time, and the three bucket groups is the SDL-Agile solution for dealing with this conundrum. However, an effect of this categorization is that teams can temporarily skip some SDL requirements for some releases. The Microsoft SDL team believes this is a necessary situation required to provide the best mix of security, feature development, and speed of release for teams with short release cycles.

Although SDL-Agile was designed for teams with short release cycles, teams with longer release cycles are still eligible to use the SDL-Agile process. However, they may find that they are actually performing more security work than if they had used the classic, waterfall-based SDL. Requirements that a team only needs to complete once in classic SDL may need to be met five or six (or more) times in SDL-Agile over the course of a long project. However, this is not necessarily a bad thing and may help the team to create a more secure product.

## Applying SDL Tasks to Sprints

While the previous section focused on requirements specific to the SDL-Agile, this section focuses on tasks associated with the SDL and how they are applied within the Agile framework.

### Security Education

Each member of a project team must complete at least one security training course every year. If more than 20 percent of the project members are out of compliance with this non-negotiable requirement, the requirement is failed (and consequently so is the sprint, and the product is not allowed to release). Consult your sprint leader for a list of courses that satisfy SDL training requirements. You can also consult the SDL Pro Network for training courses and recommendations.

Additionally, in the interests of staying lean, engineers and testers performing security-related tasks or SDL-related tasks should acquire relevant security knowledge prior to performing the tasks on the sprint. In this case, *relevant* is defined as security concepts that are pertinent to the features developed or tested during the sprint. Examples include:

***Web-based applications***

- Cross-site scripting (XSS) vulnerabilities
- SQL injection vulnerabilities

***Database applications***

- SQL injection vulnerabilities

***C and C++ applications***

- Buffer overflows
- Integer overflows

***All languages***

- Input validation
- Language-specific issues (PHP, Java, C#)

***Cryptographic code***

- Common cryptographic errors

Acquiring security knowledge could be as simple as reading appropriate chapters in a book[1] or watching an online training class. If someone on the team wants to adopt the role of "security champion" or

[1] *19 Deadly Sins of Software Security* by Howard, LeBlanc, and Viega is a book that focuses on language and domain-specific coding vulnerabilities.

security expert for their team, they should attend broader and deeper security education as part of their normal ongoing education. Having a security expert close by is advantageous to the team and, more importantly, to the customer.

## Tooling and Automation

Tools that automate security-related tasks are critical to a successful security process because the more you can automate the work necessary to meet requirements, the easier security becomes. Also, tools help reduce some of the development effort required of the developers by shifting it onto the tools. When security is involved, tools are not a replacement for humans, but tools do offer scalability—a tool can scan lots of code or check binaries without getting tired. Keep in mind, however, that simply running tools does not make a software product secure.

SDL-Agile requires the following tools to be run at least once per sprint and recommends that they be run daily or as part of the build and check-in process:

***.NET code:***

- CAT.NET (applies to ASP.NET applications only)
- FxCop 1.35 or later (all security rules at a minimum)

***Native code:***

- PREFast (or /analyze in Microsoft Visual Studio®)

## Threat Modeling: The Cornerstone of the SDL

At some point, the major SDL artifact—the threat model—must be used as a baseline for the product. Whether this is a new product or a product already under development, a threat model must be built as part of the sprint design work. Like many good Agile practices, the threat model process should be time-boxed and limited to only the parts of the product that currently exist or are in development.

Once a threat model baseline is in place, any extra work updating the threat model will usually be small, incremental changes.

A threat model is a critical part of securing a product because a good threat model helps to:

- Determine potential security design issues.
- Drive attack surface analysis and most "at-risk" components.
- Drive the fuzz-testing process.

During each sprint, the threat model should be updated to represent any new features or functionality added during that sprint. The threat model should also be updated to represent any significant design changes, even if the functionality stays the same.

### *SDL Threat Modeling Tool*

While not officially required as part of the SDL (either SDL-Agile or SDL-Classic), many internal Microsoft teams use the SDL Threat Modeling Tool with great success. The SDL Threat Modeling Tool is specifically designed to be used by developers and architects who may not necessarily have security expertise. A full review of the SDL Threat Modeling Tool is beyond the scope of this paper, but you can read more about it (and download it for free) at the Microsoft SDL Threat Modeling Tool Web site.

### *Starting a Threat Model for an Existing Project*

If an Agile team adopts the SDL-Agile as outlined in this document while a product is already in development, a threat model needs to be built for the current product, but it is imperative that the team remains lean. A minimal, but useful, threat model can be built by analyzing high-risk entry points and data in the system. At a minimum, the following should be identified and threat models built around the entry points and data:

- Anonymous and remote network endpoints
- Anonymous or authenticated local endpoints into high-privileged processes
- Sensitive, confidential, or personally identifiable data held in data stores used in the application

### *Continuing Threat Modeling*

Threat modeling is one of the every-sprint SDL requirements for SDL-Agile. Unlike most of the other every-sprint requirements, threat modeling is not easily automated and can require significant team effort. However, in keeping with the spirit of agile development, only new features or changes being implemented in the current sprint need to be threat modeled in the current sprint. This helps to minimize the amount of developer time required while still providing all the benefits of threat modeling.

## Fuzz Testing

Fuzz testing is a brutally effective security testing technique, especially if the team has never used fuzz testing on the product. The threat model should determine what portions of the application to fuzz test. If no threat model exists, the initial list should include high-risk items, such as those defined in Appendix D: High-Risk Code.

After this list is complete, the relative exposure of each entry point should be determined, and this drives the order in which entry points are fuzzed. For example, remotely accessible or unauthenticated endpoints are higher risk than local-only or authenticated endpoints.

The beauty of fuzz testing is that once a computer or group of computers is configured to fuzz the application, it can be left running, and only crashes need to be analyzed. If there are no crashes from the outset of fuzz testing, the fuzz test is probably inadequate, and a new task should be created to analyze why the fuzz tests are failing and make the necessary adjustments.

## Using a Spike to Analyze and Measure Unsecure Code in Bug Dense and "At-Risk" Code

A critical indicator of potential security bug density is the age of the code. Based on the experiences of Microsoft developers and testers, the older the code, the higher the number of security bugs found in the code. If your project has a large amount of legacy code or risky code (see Appendix D: High-Risk Code), you should locate as many vulnerabilities in this code as possible. This is achieved through a *spike*. A spike is a time-boxed "side project" with a well-defined goal (in this case, to find security bugs). You can think of this spike as a mini security push. The goal of the security push at Microsoft is to bring risky code up to date in a short amount of time relative to the project duration.

Note that the security push doesn't propose fixing the bugs yet but rather analyzing them to determine how bad they are. If a lot of security bugs are found in code with network connections or in code that handles sensitive data, these bugs should not only be fixed soon, but also another spike should be set up to comb the code more thoroughly for more security bugs.

Examples of analysis performed during a spike include:

- **All code.** Search for input validation failures leading to buffer overruns and integer overruns. Also, search for insecure passwords and key handling, along with weak cryptographic algorithms.
- **Web code.** Search for vulnerabilities caused through improper validation of user input, such as CSS.
- **Database code.** Search for SQL injection vulnerabilities.
- **Safe for scripting ActiveX controls.** Review for C/C++ errors, information leakage, and dangerous operations.

All appropriate analysis tools available to the team should be run during the spike, and all bugs triaged and logged. Critical security bugs, such as a buffer overrun in a networked component or a SQL injection vulnerability, should be treated as high-priority *unplanned items*.

## Exceptions

The SDL requirement exception workflow is somewhat different in SDL-Agile than in the classic SDL. Exceptions in SDL-Classic are granted for the life of the release, but this won't work for Agile projects. A "release" of an Agile project may only last for a few days until the next sprint is complete, and it would be a waste of time for project managers to keep renewing exceptions every week.

To address this issue, project teams following SDL-Agile can choose to either apply for an exception for the duration of the sprint (which works well for longer sprints) or for a specific amount of time, not to exceed six months (which works well for shorter sprints). When reviewing the requirement exception, the security advisor can choose to increase or decrease the severity of the exception by one level (and thus increase or decrease the seniority of the manager required to approve the exception) based on the requested exception duration.

For example, say a team requests an exception for a requirement normally classified as severity 3, which requires manager approval. If they request the exception only for a very short period of time, say two weeks, the security advisor may drop the severity to a 4, which requires only approval from the team's security champion. On the other hand, if the team requests the full six months, the security advisor may increase the severity to a 2 and require signoff from senior management due to the increased risk.

In addition to applying for exceptions for specific requirements, teams can also request an exception for an entire bucket. Normally teams must complete at least one requirement from each of the bucket categories during each sprint, but if a team cannot complete even one requirement from a bucket, the team requests an exception to cover that entire bucket. The team can request an exception for the duration of the sprint or for a specific time period, not to exceed six months, just like for single exceptions. However, due to the broad nature of the exception—basically stating that the team is going to skip an entire category of requirements—bucket exceptions are classified as severity 2 and require the approval of at least a senior manager.

## Final Security Review

A Final Security Review (FSR) similar to the FSR performed in the classic waterfall SDL is required at the end of every agile sprint. However, the SDL-Agile FSR is limited in scope—the security advisor only needs to review the following:

- All every-sprint requirements have been completed, or exceptions for those requirements have been granted.

- At least one requirement from each bucket requirement category has been completed (or an exception has been granted for that bucket).
- No bucket requirement has gone more than six months without being completed (or an exception has been granted).
- No one-time requirements have exceeded their grace period deadline (or exceptions have been granted).
- No security bugs are open that fall above the designated severity threshold (that is, the security bug bar).

Some of these tasks may require manual effort from the security advisor to ensure that they have been completed satisfactorily (for example, threat models should be reviewed), but in general, the SDL-Agile FSR is considerably more lightweight than the SDL-Classic FSR.

Now that the basic methodology and foundation is in place, it's time for an example scenario.

## SDL-Agile Example

A database-driven Web product is currently in development by a team with a four-week sprint duration. It is primarily written using C# and ASP.NET. There is a Windows service that processes some data from the Web application. The service was originally written three years ago and is about 11,000 lines of C++ code—it's pretty complex.

Input to the Web application is mostly unauthenticated, but it does offer a remotely accessible admin-only interface. The application also uses a small ActiveX control written in C++.

The product backlog includes 45 user stories—21 of these are high-priority stories, 10 are medium-priority stories, and 14 are low-priority stories. During the sprint planning phase, 10 user stories are selected for the current sprint, 3 stories are high priority, 3 are medium priority, and the final 4 are low priority.

At this point, the team adds technology stories for each of the every-sprint SDL requirements. Even though the product uses both managed and native modules, the team is only working on the managed-code modules during this sprint, so only the every-sprint tasks that apply to managed online services are added to the sprint.

Since this is the first sprint in which the team is using the SDL-Agile process, additional high-priority stories are added to complete some of the one-time requirements (for example, registering the project in the security compliance tracking system, creating a privacy form, identifying a privacy incident response person, and identifying a security program manager). One more high-priority story is added to update the build process to integrate the SDL-required, every-build requirements (use of the SDL-required compiler and linker flags and integration of the FxCop security rules).

Finally, the team also adds in high-priority stories for the bucket tasks that the team wants to complete during the current sprint. For this sprint, the team chooses to add tasks to run an attack surface analyzer, review the crypto design of the system, and create a content publishing and user interface security plan.

The sprint begins, and two people on the team take on the task of building the threat model for the features to be developed during this sprint. The big problem is that no one knows how to build a threat model, so the two people read the threat modeling chapter in the SDL book[2] and read Adam Shostack's series of threat modeling blog posts. This gives them enough information to perform the threat modeling task.

After the threat model is built (and the corresponding story completed), the team uncovers a critical vulnerability—the database contains sensitive data (users' names, computer information, browser information, and IP addresses), and the data is not protected from disclosure threats. Because the data is sensitive, and it appears that unauthenticated attackers could access the data through a potential SQL injection vulnerability, two more high-priority stories are added to the sprint backlog—one to add defenses to protect the data in the database and the other to scour the code for SQL injection vulnerabilities. The team does not know about protecting data from disclosure, so the person slated to work on this defense reads chapter 12, "Failing to Store and Protect Data Securely," in the *19 Deadly Sins*.

One developer checks to ensure that the build environment is set up to use the SDL-required compiler and linker switches and that the security-focused code analysis tools are also set to run as part of the build process.

After looking at the new set of user stories so far, the team decides to remove two medium-priority stories and two low-priority stories to keep within the sprint time box—these stories are put back in the product backlog. Finally, after talking to the customer, one high-priority story is downgraded to a medium-priority story but is to be completed in this sprint.

One developer elects to address the possible SQL injection vulnerabilities identified by the threat model. He spends three days finding all database access code within the Web and C++ code and modifying it to use stored procedures and parameterized queries. He also modifies the access rights of the interactive database user so that it does not have access to any database tables that are not necessary for the application. He also removes the interactive user's permissions for deleting database objects and creating new database objects, since these are also not necessary for the application to function. These practices are good defense-in-depth measures that help prevent the system from being exploited in the event that a vulnerability accidentally slips into the production code.

Nineteen days into the sprint, all of the SDL-required, high-priority stories are completed, as are many of the selected user stories. The team finishes out the sprint, completing the rest of the selected user stories. The sprint is a success, and the team is poised to release their new code to the public.

[2] Howard, Michael, and Steve Lipner. *The Security Development Lifecycle* (Chapter 9). Microsoft Press, June 28, 2006.

## Appendix A: Every-Sprint Requirements

| Title | Requirement/ Recommendation | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|
| **Communicate privacy-impacting design changes to the team's privacy advisor** | Requirement | X | X | X |
| **Compile all code with the /GS compiler option** | Requirement | X | | X |
| **Comply with SDL firewall requirements** | Requirement | | X | X |
| **Do not use banned APIs in new code** | Requirement | X | | X |
| **Ensure all ASP.NET applications use the ValidateRequest cross-site scripting input validation attribute** | Requirement | X | X | |
| **Ensure all database access is performed through parameterized queries to stored procedures** | Requirement | X | X | X |
| **Ensure all team members have had security education within the past year** | Requirement | X | X | X |
| **Ensure the application domain group is granted only execute permissions on the database stored procedures** | Requirement | X | X | X |
| **Fix all issues identified by code analysis tools for unmanaged code** | Requirement | X | | X |
| **Fix all security issues identified by CAT.NET and FxCop static analysis** | Requirement | X | X | |
| **Follow input validation and output encoding guidelines to defend against cross-site scripting attacks** | Requirement | X | X | X |

| Title | Requirement/ Recommendation | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|
| **Link all code with the /dynamicbase linker option (Address Space Layout Randomization)** | Requirement | X | | X |
| **Link all code with the /nxcompat linker option (Data Execution Prevention)** | Requirement | | | X |
| **Link all code with the /safeseh linker option (safe exception handling)** | Requirement | | | X |
| **Update threat models for new features** | Requirement | X | X | X |
| **Use HeapSetInformation** | Requirement | | | X |
| **Use the /robust MIDL compiler switch** | Requirement | | | X |
| **Use the Relying Party Suite SDK** | Requirement | | X | X |
| **Avoid JavaScript eval function and equivalents** | Recommendation | X | | |
| **Canonicalize URLs** | Recommendation | X | X | X |
| **Encode long-lived pointers** | Recommendation | X | | X |
| **Review error messages to ensure sensitive information is not disclosed** | Recommendation | X | X | X |
| **Use standard annotation language (SAL) to annotate all functions** | Recommendation | X | | X |
| **Use strict /GS option** | Recommendation | X | | X |
| **Use whitelist of allowed domains to perform redirects** | Recommendation | X | X | X |

# Appendix B: Bucket Requirements

## Bucket A: Security Verification

| Title | Requirement/ Recommendation | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|
| **Debug the application with the Application Verifier enabled** | Requirement | | | X |
| **Disable tracing and debugging in ASP.NET applications** | Requirement | X | X | |
| **Investigate and service any reported /GS crashes** | Requirement | | | X |
| **Perform ActiveX control fuzzing** | Requirement | X | | X |
| **Perform attack surface analysis** | Requirement | X | X | X |
| **Perform binary analysis (BinScope)** | Requirement | X | X | X |
| **Perform COM object testing** | Requirement | | | X |
| **Perform cross-domain scripting testing** | Requirement | X | X | X |
| **Perform file fuzz testing** | Requirement | X | | X |
| **Perform RPC fuzz testing** | Requirement | X | | X |
| **Conduct in-depth manual and automated code review for high-risk code** | Recommendation | X | X | X |
| **Perform data flow testing** | Recommendation | X | X | X |
| **Perform input validation testing** | Recommendation | X | X | X |
| **Perform replay testing** | Recommendation | X | X | X |

## Bucket B: Design Review

| Title | Requirement/ Recommendation | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|
| **Avoid cross-domain access to authenticated sites** | Requirement | X | X | X |
| **Comply with User Account Control (UAC) best practices to ensure all code runs as a non-administrator** | Requirement | | X | X |
| **Conduct a privacy review** | Requirement | X | X | X |
| **Ensure all code is compliant with the SDL Cryptographic Standards** | Requirement | X | X | X |
| **Ensure all code is compliant with the SDL Privacy Guidelines document** | Requirement | X | X | X |
| **Use strongly named assemblies, and request minimal permissions** | Requirement | X | X | |
| **Complete in-depth threat model training** | Recommendation | X | X | X |
| **Disable rarely used features by default, to reduce attack surface** | Recommendation | X | X | X |
| **Grant minimal privileges** | Recommendation | X | X | X |
| **Review planning and design specifications for user interface elements** | Recommendation | X | X | X |
| **Use Windows Imaging Component to process image data** | Recommendation | X | | X |

## Bucket C: Response Plans

| Title | Requirement/ Recommendation | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|
| **Add or update privacy scenarios in the test plan** | Requirement | X | X | X |
| **Create or update the list of response contacts** | Requirement | X | X | X |
| **Define or update the privacy bug bar** | Requirement | X | X | X |
| **Define or update the security bug bar** | Requirement | X | X | X |
| **Ensure symbols are available internally for all public releases** | Requirement | X | X | X |
| **Create or update a business continuity-disaster recovery plan** | Recommendation | X | X | X |
| **Create or update a network down plan** | Recommendation | X | X | X |
| **Create or update content publishing plan** | Recommendation | X | X | X |
| **Create or update privacy support documents** | Recommendation | X | X | X |

## Appendix C: SDL-Agile One-Time Requirements

| Title | Requirement/ Recommendation | Completion Deadline (months) | Applies to Online Services | Applies to Managed Code | Applies to Native Code |
|---|---|---|---|---|---|
| **Avoid writable PE segments** | Requirement | 6 | X | | X |
| **Create a baseline threat model** | Requirement | 3 | X | X | X |
| **Determine security response standards** | Requirement | 6 | X | X | X |
| **Establish a security response plan** | Requirement | 6 | X | X | X |
| **Identify primary security and privacy contacts** | Requirement | 1 | X | X | X |
| **Identify your team's privacy expert** | Requirement | 1 | X | X | X |
| **Identify your team's security expert** | Requirement | 1 | X | X | X |
| **Use approved XML parsers** | Requirement | 6 | X | | X |
| **Use latest compiler versions** | Requirement | 12 | X | X | X |
| **Configure bug tracking to track the cause and effect of security bugs** | Recommendation | 3 | X | X | X |
| **Designate full-time security program manager** | Recommendation | 3 | X | X | X |
| **Remove dependencies on NTLM authentication** | Recommendation | 12 | X | X | X |

## Appendix D: High-Risk Code

The following defines the highest risk code (at the time of writing) that should receive greater scrutiny if the code is legacy code and should be written with the greatest care if the code is new code.

- Windows services and *nix daemons listening on network connections
- Windows services running as SYSTEM or *nix daemons running as root
- Code listening on unauthenticated network ports connections
- ActiveX controls
- Browser protocol handlers (for example, about: or mms:)
- setuid root applications on *nix
- Code that parses data from untrusted (non-admin or remote) files
- File parsers or MIME handlers

## Appendix E: Frequently Asked Questions

**Q:** Can teams release products without ever having to complete some requirements?

**A:** Yes, but it is not the intent of SDL-Agile to allow teams to ignore or avoid certain SDL requirements indefinitely. This is a side effect of a process that is designed to respect the needs of the team to spend a significant amount of time innovating and implementing new features while still maintaining an appropriate security baseline. No requirement can go more than six months without being completed (or having an exception granted).

**Q:** Why not mandate a round-robin or other type of requirement rotation to ensure that all requirements eventually get addressed?

**A:** Some teams feel strongly that certain requirements are a better use of their limited time budget. If, for example, a team feels that the process of running and analyzing attack surface analyzer results is not as valuable as running and analyzing file fuzzer results, it can perform file fuzzing more often and attack surface analysis less often.

**Q:** Why not mandate a security spike—a sprint totally focused on security?

**A:** If teams want to do this, great! But it is not part of the SDL-Agile requirements. In general, one of the guiding principles of SDL-Agile is to keep teams from spending so much time on security that it significantly affects their feature velocity. A mandated security spike would definitely affect a team's feature release schedule.