Microsoft | Security Development Lifecycle

SDL Process Guidance Version 5.2

May 23, 2012

Microsoft

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

This documentation is not an exhaustive reference on the SDL process as practiced at Microsoft. Additional assurance work may be performed by product teams (but not necessarily documented) at their discretion. As a result, this example should not be considered as the exact process that Microsoft follows to secure all products.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

Microsoft Security Development Lifecycle (SDL) is an industry-leading software security assurance process. A Microsoft-wide initiative and a mandatory policy since 2004, the SDL has played a critical role in embedding security and privacy in Microsoft software and culture. Combining a holistic and practical approach, the SDL introduces security and privacy early and throughout all phases of the development process. It has led Microsoft to measurable and widely-recognized security improvements in flagship products such as Windows Vista and SQL Server. Microsoft is publishing its detailed SDL process guidance to provide transparency on the secure software development process used to develop its products.

The following documentation provides an in-depth description of the Microsoft SDL methodology and requirements used at Microsoft. Proprietary technologies and resources that are only available internally at Microsoft have been omitted from these guidelines.

Organizations that wish to implement the SDL should read the Simplified Implementation of the Microsoft SDL whitepaper. This whitepaper illustrates the core concepts of the Microsoft SDL and discusses the individual security activities that should be performed in order to follow the SDL process. Visit www.microsoft.com/sdl for resources and tools.

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

Contents

Changes in This Version

Corrected typographical errors and added guidance regarding SDL security requirements and security recommendations.

- Phase Two: Design
  - One new security requirement
  - Two updated security requirements
  - One new security recommendation
  - One updated security recommendation

- Phase Three: Implementation
  - Two security requirements promoted from recommendations

- SDL for Line-of-Business Applications
  - Two new security requirements
  - One updated security requirements

- Appendix N
  - Updated guidance

Introduction

All software developers must address security threats. Computer users now require trustworthy and secure software, and developers who address security threats more effectively than others can gain a competitive advantage in the marketplace. Also, an increased sense of social responsibility now compels developers to create secure software that requires fewer patches and less security management.

Privacy also demands attention. To ignore privacy concerns of users can invite blocked deployments, litigation, negative media coverage, and mistrust. Developers who protect privacy earn users' loyalties and distinguish themselves from their competitors.

Secure software development has three elements—best practices, process improvements, and metrics. This document focuses primarily on the first two elements, and metrics are derived from measuring how they are applied.

Microsoft has implemented a stringent software development process that focuses on these elements. The goal is to minimize security-related vulnerabilities in the design, code, and documentation and to detect and eliminate vulnerabilities as early as possible in the development life cycle. These improvements reduce the number and severity of security vulnerabilities and improve the protection of users' privacy.

Secure software development is mandatory for software that is developed for the following uses:

- In a business environment
- To process personally identifiable information (PII) or other sensitive information
- To communicate regularly over the Internet or other networks

(For more specific definitions, see What Products and Services Are Required to Adopt the Security Development Lifecycle Process? later in this introduction.)

This document describes both required and recommended changes to software development tools and processes. These changes should be integrated into existing software development processes to facilitate best practices and achieve measurably improved security and privacy.

Note: This document outlines the SDL process used by Microsoft product groups for application development. It has been modified slightly to remove references to internal Microsoft resources and to minimize Microsoft-specific jargon. We make no guarantees as to its applicability for all types of application development or for all development environments. Implementers should use common sense in choosing the portions of the SDL that make sense given existing resources and management support.

Traditional Microsoft Product Development Process

In response to the Trustworthy Computing (TwC) directive of January 2002, many software development groups at Microsoft instigated *security pushes* to find ways to improve the security of existing code and one or two prior versions of the code. However, the reliable delivery of more secure software requires a comprehensive process, so Microsoft defined *Secure by Design, Secure by Default, Secure in Deployment, and Communications* (SD3+C) to help determine where security and privacy efforts are needed. The guiding principles for SD3+C are identified in the following subsections.

Secure by Design

- **Secure architecture, design, and structure.** Developers consider security issues part of the basic architectural design of software development. They review detailed designs for possible security issues and design and develop mitigations for all threats.
- **Threat modeling and mitigation.** Threat models are created, and threat mitigations are present in all design and functional specifications.
- **Elimination of vulnerabilities.** No known security vulnerabilities that would present a significant risk to anticipated use of the software remain in the code after review. This review includes the use of analysis and testing tools to eliminate classes of vulnerabilities.
- **Improvements in security.** Less secure legacy protocols and code are deprecated, and, where possible, users are provided with secure alternatives that are consistent with industry standards.

Secure by Default

- **Least privilege.** All components run with the fewest possible permissions.
- **Defense in depth.** Components do not rely on a single threat mitigation solution that leaves users exposed if it fails.
- **Conservative default settings.** The development team is aware of the attack surface for the product and minimizes it in the default configuration.
- **Avoidance of risky default changes.** Applications do not make any default changes to the operating system or security settings that reduce security for the host computer. In some cases, such as for security products (for example, Microsoft Internet Security and Acceleration [ISA] Server), it is acceptable for a software program to strengthen (increase) security settings for the host computer. The most common violations of this principle are games that either open up firewall ports without informing the user or instruct users to do so without consideration of possible risks.
- **Less commonly used services off by default.** If fewer than 80 percent of a program's users use a feature, that feature should not be activated by default. Measuring 80 percent usage in a product is often difficult because programs are designed for many different personas. It can be useful to

consider whether a feature addresses a core/primary use scenario for all personas. If it does, the feature is sometimes referred to as a P1 feature.

Secure in Deployment

- **Deployment guides.** Prescriptive deployment guides outline how to deploy each feature of a program securely, including providing users with information that enables them to assess the security risk of activating non-default options (and thereby increasing the attack surface).
- **Analysis and management tools.** Security analysis and management tools enable administrators to determine and configure the optimal security level for a software release. These tools include Microsoft Baseline Security Analyzer and Group Policy, through which you can manage all security-related configuration options.
- **Patch deployment tools.** Deployment tools are provided to aid in patch deployment.

Communications

- **Security response.** Development teams respond promptly to reports of security vulnerabilities and communicate information about security updates.
- **Community engagement.** Development teams proactively engage with users to answer questions about security vulnerabilities, security updates, or changes in the security landscape.

An analogous concept to SD3+C for privacy is known as PD3+C. The guiding principles for PD3+C are outlined in the following subsections.

Privacy by Design

- **Provide notice and consent.** Provide appropriate notice about data that is collected, stored, or shared so that users can make informed decisions about their personal information.
- **Enable user policy and control.** Enable parents to manage privacy settings for their children, and enable enterprises to manage privacy settings for their employees.
- **Minimize data collection and sensitivity.** Collect the minimum amount of data that is required for a particular purpose, and use the least sensitive form of that data.
- **Protect the storage and transfer of data.** Encrypt PII in transfer, limit access to stored data, and ensure that data usage complies with uses communicated to the user.

Privacy by Default

- **Ship with conservative default settings.** Obtain appropriate consent before collecting or transferring any data. To prevent unauthorized access, protect personal data stored in access control lists.

Privacy in Deployment

- **Publish deployment guides.** Disclose privacy mechanisms to enterprise users so that they can establish internal privacy policies and maintain their users' and employees' privacy.

Communications

- **Publish author-appropriate privacy disclosures.** Post privacy statements on appropriate websites.
- **Promote transparency.** Actively engage mainstream and trade media outlets with white papers and other documentation to help reduce anxiety about high-risk features.
- **Establish a privacy response team.** Assign staff responsible for responding if a privacy incident or escalation occurs.

Security Development Lifecycle (SDL)

After you add steps to the development process for all elements of SD3+C, the secure software development process model looks like the one shown in Figure 1.

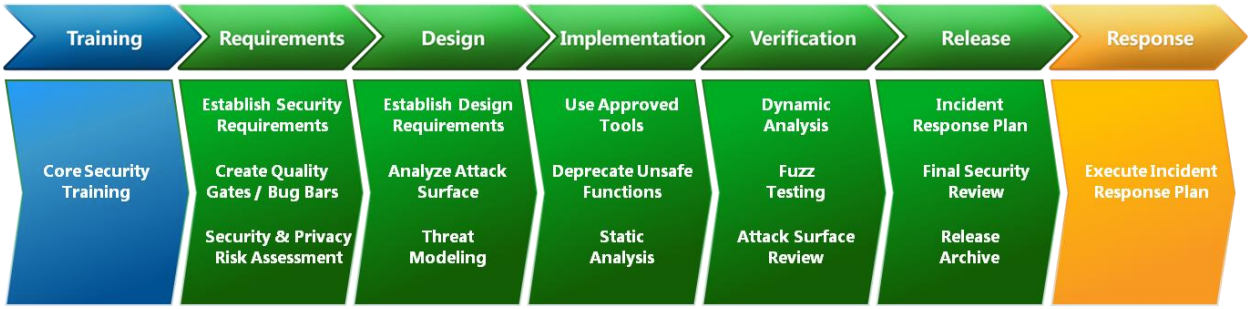

Figure 1. Secure software development process model at Microsoft

Process improvements are incremental and do not require radical changes in the development process. However, it is important to make improvements consistently across an organization.

The rest of this document describes each step of the process in detail.

What Products and Services Are Required to Adopt the SDL Process?

- Any software release that is commonly used or deployed within any organization, such as a business organization or a government or nonprofit agency.
- Any software release that regularly stores, processes, or communicates PII or other sensitive information. Examples include financial or medical information.
- Any software product or service that targets or is attractive to children 13 years old or younger.
- Any software release that regularly connects to the Internet or other networks. Such software might be designed to connect in different ways, including:
  - **Always online.** Services provided by a product that involve a presence on the Internet (for example, Windows® Messenger).
  - **Designed to be online.** Browser or mail applications that expose Internet functionality (for example, Microsoft Office Outlook® or Microsoft Internet Explorer®).
  - **Exposed online.** Components that are routinely accessible through other products that interact with the Internet (for example, Microsoft ActiveX® controls or PC–based games with multiplayer online support).

- Any software release that automatically downloads updates.
- Any software release that accepts or processes data from an unauthenticated source, including:
  - Callable interfaces that "listen."
  - Functionality that parses any unprotected file types that should be limited to system administrators.
- Any release that contains ActiveX controls.
- Any release that contains COM controls.

Are Service Releases Required to Adopt the SDL Process?

Any external release of software that can be installed on a user's computer, regardless of operating system or platform, must comply with security and privacy policies as described in the Security Development Lifecycle. The SDL applies to new products, service releases such as product service packs, feature packs, development kits, and resource kits. The terms *service pack* and *feature pack* might not always be used in the descriptive title of a release to users, but the following definitions differentiate what constitutes a *new product* from a service release or feature pack.

- *New product releases* are either completely new products (version 1.0) or significant updates of existing products (for example, Microsoft Office 2003). A new product release always requires a user to agree to a new software license and typically involves new packaging.
- *Service releases* include service packs or feature packs and resource kits.
- *Service packs* are the means by which product updates are distributed. Service packs might contain updates for system reliability, program compatibility, security, or privacy. A service pack requires a previous version of a product before it can be installed and used. A service pack might not always be named as such; some products may refer to a service pack as a *service release*, *update*, or *refresh*.
- *Resource kits* are collections of resources to help administrators streamline management tasks. A resource kit must be targeted at a single product release to be treated as a service release. If a resource kit is targeted at multiple products or at multiple versions of a product, SDL requirements apply to it as described earlier for a product release.
- *Development kits* provide information, specific architecture details, and tools to developers. A development kit must be targeted at a single product release to be treated as a service release. If a development kit is targeted at multiple products or at multiple versions of a product, SDL requirements from the corresponding product release apply.

All software releases referenced in What Products and Services Are Required to Adopt the SDL Process? must adopt the SDL. However, current SDL requirements are applied only to the new features in the service release and not retroactively to the entire product. Also, product teams are not required to change compiler versions or compile options in a service release.

How Are New Recommendations and Requirements Added to the SDL Process?

The Security Development Lifecycle consists of the proven best practices and tools that were successfully used to develop recent products. However, the area of security and privacy changes frequently, and the Security Development Lifecycle must continue to evolve and to use new knowledge and tools to help build even more trusted products. But because product development teams must also have some visibility and predictability of security requirements in order to plan schedules, it is necessary to define how new recommendations and requirements are introduced, as well as when new requirements are added to the SDL.

New SDL recommendations may be added at any time, and they do not require immediate implementation by product teams. New SDL requirements should be released and published at six-month intervals. New requirements will be finalized and published three months before the beginning of the next six-month interval for which they are required. For more information about how to hold teams accountable for requirements, see How Are SDL Requirements Determined for a Specific Product Release?

The list of required development tools (for example, compiler versions or updated security tools) is typically the area of greatest interest because of the potential impact on schedule and resources. The following example timeline helps to illustrate this point:

- **October 1, 2012.** Publish updated requirements that will apply to all products registering after January 1, 2012.
- **January 1, 2013.** Updated requirements list takes effect.
- **April 1, 2013.** Publish updated requirements that will apply to all products registering after July 1, 2012.
- **July 1, 2013.** Updated requirements list takes effect.

How Are SDL Requirements Determined for a Specific Product Release?

A product release is held accountable for the SDL requirements that are current on the day the product registers a request for SDL review. Product teams can refer to the SDL version numbers to determine the appropriate policies to follow. There are some caveats to this rule:

- **One-year cap.** At a minimum, a product must meet SDL requirements that are older than one year at the time of release to manufacture (RTM) or release to web (RTW).
- **Multi-version limit.** If you register more than one version of your product to be released in succession, later versions are held to the requirements that are in effect on the date the previous version was released.
- **Previous version.** All projects that are already registered before July 1 of a given year are subject to the SDL requirements published on January 1 of the same year.

- **Threat evolution.** The security engineering team reserves the right to add new requirements at any time in response to the availability of high-impact tools or the evolution of new threats.

The following examples illustrate how SDL requirements are determined:

- If a product registered with the SDL team in the first half of calendar year 2012 (H1CY12) but does not ship until H2CY13, it must meet all H2CY12 requirements.
- If a team registers two versions of a product to be released within a three-month period, the first version is subject to current requirements, but the second version is subject to the published requirements on the day the first version ships.

Pre-SDL Requirements: Security Training

Education and Awareness

All members of software development teams should receive appropriate training to stay informed about security basics and recent trends in security and privacy. Individuals who develop software programs should attend at least one security training class each year. Security training can help ensure software is created with security and privacy in mind and can also help development teams stay current on security issues. Project team members are strongly encouraged to seek *additional* security and privacy education that is appropriate to their needs or products.

A number of key knowledge concepts are important to successful software security. These concepts can be broadly categorized as either basic or advanced security knowledge. Each technical member of a project team (developer, tester, program manager) should be exposed to the knowledge concepts in the following subsections.

Basic Concepts

- **Secure design**, including the following topics:
  - Attack surface reduction
  - Defense in depth
  - Principle of least privilege
  - Secure defaults
- **Threat modeling**, including the following topics:
  - Overview of threat modeling

    - Design to a threat model
    - Coding to a threat model
    - Testing to a threat model
- **Secure coding**, including the following topics:
    - Buffer overruns
    - Integer arithmetic errors
    - Cross-site scripting
    - SQL injection
    - Weak cryptography
    - Managed code issues (Microsoft .NET/Java)
- **Security testing**, including the following topics:
    - Security testing versus functional testing
    - Risk assessment
    - Test methodologies
    - Test automation
- **Privacy**, including the following topics:
    - Types of privacy data
    - Privacy design best practices
    - Risk analysis
    - Privacy development best practices
    - Privacy testing best practices

Advanced Concepts

The preceding training concepts establish an adequate knowledge baseline for technical personnel. As time and resources permit, it is recommended that you explore other advanced concepts. Examples include (but are not limited to):

- Security design and architecture.
- User interface design.
- Security concerns in detail.
- Security response processes.
- Implementing custom threat mitigations.

Security Requirements

- All developers, testers, and program managers must complete at least one security training class each year. Individuals who have not taken a class in the basics of security design, development, and testing must do so.
- At least 80 percent of the project team staff who work on products or services must be in compliance with the standards listed earlier before their product or service is released. Relevant managers must also be in compliance with these standards. Project teams are strongly encouraged to plan security training early in the development process so that training can be completed as early as possible and have a maximum positive effect on the project's security.

Security Recommendations

Microsoft recommends that staff who work in all disciplines read the following publications:

- *Writing Secure Code, Second Edition* (ISBN 9780735617223; ISBN: 0-7356-1722-8).
- *Uncover Security Design Flaws Using the STRIDE Approach* (ISBN: 0-7356-1991-3).

Privacy Recommendations

Microsoft recommends that staff who work in all disciplines read the following documents:

- Appendix A: Privacy at a Glance (Sample)
- *Microsoft Privacy Guidelines for Developing Software Products and Services*

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 5: Stage 0—Education and Awareness.
- *Privacy: What Developers and IT Professionals Should Know* (ISBN-10: 0-321-22409-4; ISBN-13: 978-0-321-22409-5)
- *The Protection of Information in Computer Systems*
- *Bell-LaPadula Model*
- *Biba Model*

Phase One: Requirements

The Requirements phase of the SDL includes the project inception—when you consider security and privacy at a foundational level—and a cost analysis—when you determine if development and support costs for improving security and privacy are consistent with business needs.

Project Inception

The need to consider security and privacy at a foundational level is a fundamental tenet of system development. The best opportunity to build trusted software is during the initial planning stages of a new release or a new version because development teams can identify key objects and integrate security and privacy, which minimizes disruption to plans and schedules.

Security Requirements

- Develop and answer a short questionnaire to verify whether your development team is subject to Security Development Lifecycle (SDL) policies. The questionnaire has two possible outcomes:
  1. If the project is subject to SDL policies, it must be assigned a security advisor who serves as the point of contact for its Final Security Review (FSR). It is in the project team's interest to register promptly and establish the security requirements for which they will be held accountable. The team will also be asked some technical questions as part of a security risk assessment to help the security advisor identify potential security risks. (See Cost Analysis in this document.)
  2. If the project is not subject to SDL policies, it is not necessary to assign a security advisor and the release is classified as exempt from SDL security requirements.
- Identify the team or individual that is responsible for tracking and managing security for the product. This team or individual does not have sole responsibility for ensuring that a software release is secure, but the team or individual is responsible for coordinating and communicating the status of any security issues. In smaller product groups, a single program manager might take on this role.
- Ensure that bug reporting tools can track security issues and that a database can be queried dynamically for all security bugs at any time. The purpose of this query is to examine unfixed security issues in the FSR. The project's bug tracking system must accommodate the *bug bar* ranking value recorded with each bug.
- Define and document the project's security bug bar. This set of criteria establishes a minimum level of quality. Defining it at the start of the project improves understanding of risks associated with security issues and enables teams to identify and fix security issues during development. The project team must negotiate a bug bar approved by the security advisor with project-specific clarifications and (as appropriate) more stringent security requirements specified by the security advisor. ***The bug bar***

must never be relaxed, though, even as the project's release date nears. Bug bar examples can be found in Appendix M: SDL Privacy Bug Bar and Appendix N: SDL Security Bug Bar.

- Include third-party code licensing security requirements in all new contracts. If your product contains licensed third-party code or hardware that is newly contracted in the current release, you should ensure that there is a provision in the licensing contract that requires the third party to provide proof of compliance with a predefined list of SDL requirements. Depending upon your company's relationship with the third party, this can include things such as the source code itself for verification, security tool output and logs, or independent verification performed by an outside agent.

Privacy Requirements

- Identify the privacy advisor who will serve as your team's first point of contact for privacy support and additional resources.
- Identify the team member responsible for privacy for the project. This person is typically called the *privacy lead* or, sometimes, the *privacy champion*.
- Define and document the project's privacy bug bar (see the preceding **Security Requirements** section).

Security Recommendations

It is useful to create a security plan document during the Design phase to outline the processes and work items your team will follow to integrate security into their development process. The security plan should identify the timing and resource requirements that the Security Development Lifecycle prescribes for individual activities. These requirements should include:

- Team training.
- Threat modeling.
- Security push.
- Final Security Review (FSR).

The security plan should reflect a development team's overall perspective on security goals, challenges, and plans. Security plans can change, but articulating one early helps ensure that no requirements are overlooked and avoids last-minute surprises. A sample security plan is included in Appendix O**.**

Consider using a tool to track security issues by cause and effect. This information is very important to have later in a project. Ensure that the bug reporting tool used includes fields with the STRIDE values in the following lists (definitions for these values are available in Appendix B: Security Definitions for Vulnerability Work Item Tracking).

The tool's **Security Bug Effect** field should be set to one or more of the following STRIDE values:

- Not a Security Bug
- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation Disclosure
- **D**enial of Service
- **E**levation of Privilege
- Attack Surface Reduction

It is also important to use the **Security Bug Cause** field to log the cause of a vulnerability (this field should be mandatory if **Security Bug Effect** is anything other than *Not a Security Bug*).

The **Security Bug Cause** field should be set to one of the following values:

- Not a security bug
- Buffer overflow/underflow
- Arithmetic error (for example, integer overflow)
- SQL/Script injection
- Directory traversal
- Race condition
- Cross-site scripting
- Cryptographic weakness
- Weak authentication
- Weak authorization/Inappropriate permission or access control list (ACL)
- Ineffective secret hiding
- Unlimited resource consumption (Denial of Service [DoS])
- Incorrect/No error messages
- Incorrect/No pathname canonicalization
- Other

Be sure to configure bug reporting tools correctly; limit access to bugs with security implications to the project team and security advisors only.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 6: Stage 1—Project Inception.

Cost Analysis

Before you invest time in design and implementation, it is important to understand the costs and requirements involved in handling data with privacy considerations. Privacy risks increase development and support costs, so improving security and privacy can be consistent with business needs.

Security Requirements

A security risk assessment (SRA) is a mandatory exercise to identify functional aspects of the software that might require deep security review. Given that program features and intended functionality might be different from project to project, it is wise to start with a simple SRA and expand it as necessary to meet the project scope.

Such assessments must include the following information:

- What portions of the project will require threat models before release.
- What portions of the project will require security design reviews before release.
- What portions of the project will require penetration testing (*pen testing*) by a mutually agreed-upon group that is external to the project team. Any portion of the project that requires pen testing must resolve issues identified during pen testing before it is approved for release.
- Any additional testing or analysis requirements the security advisor deems necessary to mitigate security risks.
- Clarification of the specific scope of *fuzz testing* requirements. (Verification Phase: Security and Privacy Testing discusses fuzz testing.)

Note: SRA guidelines are discussed in Chapter 8 of *The Security Development Lifecycle*, along with a sample SRA on the DVD included with the book.

Privacy Requirements

Complete the Initial Assessment of the Appendix C: SDL Privacy Questionnaire. An initial assessment is a quick way to determine a project's Privacy Impact Rating and to estimate how much work is necessary to comply with Microsoft Privacy Guidelines for Developing Software Products and Services.

The Privacy Impact Rating (P1, P2, or P3) measures the sensitivity of the data your software will process from a privacy point of view. More information about Privacy Impact Ratings can be found in Chapter 8 of *The Security Development Lifecycle*. General definitions of privacy impact are defined as:

- **P1 High Privacy Risk.** The feature, product, or service stores or transfers PII or error reports, monitors the user with an ongoing transfer of anonymous data, changes settings or file type associations, or installs software.

- **P2 Moderate Privacy Risk.** The sole behavior that affects privacy in the feature, product, or service is a one-time, user-initiated, anonymous data transfer (for example, the user clicks a link and goes out to a website).
- **P3 Low Privacy Risk.** No behaviors exist within the feature, product, or service that affect privacy. No anonymous or personal data is transferred, no PII is stored on the machine, no settings are changed on the user's behalf, and no software is installed.

Product teams must complete only the work that is relevant to their Privacy Impact Rating. Complete the initial assessment early in the product planning/requirements phase, before you write detailed specifications or code.

Privacy Recommendations

If your Privacy Impact Rating is P1 or P2, understand your obligations and try to reduce your risk. Early awareness of all the required steps for deploying a project with high privacy risk might help you decide whether the costs are worth the business value gained. Review the guidance in Understand Your Obligations and Try to Lower Your Risk of Appendix C: SDL Privacy Questionnaire. If your Privacy Impact Rating is P1, schedule a "sanity check" with your organization's privacy expert. This person should be able to guide you through implementation of a high-risk project and might have other ideas to help you reduce your risk.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 7: Stage 2—Define and Follow Design Best Practices
- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 8: Stage 3—Product Risk Assessment
- Microsoft Privacy Guidelines for Developing Software Products and Services

Phase Two: Design

The Design phase is when you build the plan for how you will take your project through the rest of the SDL process—from implementation, to verification, to release. During the Design phase you establish best practices to follow for this phase by way of functional and design specifications, and you perform risk analysis to identify threats and vulnerabilities in your software.

Establish and Follow Best Practices for Design

The best time to influence a project's trustworthy design is early in its life cycle. Functional specifications may need to describe security features or privacy features that are directly exposed to users, such as requiring user authentication to access specific data or user consent before use of a high-risk privacy feature. Design specifications should describe how to implement these features and how to implement all functionality as secure features. *Secure features* are defined as features with functionality that is well engineered with respect to security, such as rigorously validating all data before processing it or cryptographically robust use of cryptographic APIs. It is important to consider security and privacy concerns carefully and early when you design features and to avoid attempts to add security and privacy near the end of a project's development.

Threat modeling (described in Design Phase: Risk Analysis) is the other critical security activity that must be completed during the design phase.

Security Requirements

- Complete a security design review with a security advisor for any project or portion of a project that requires one. Some low-risk components might not require a detailed security design review.

  The AllowPartiallyTrustedCallersAttribute (APTCA) enables an assembly to be called by untrusted code. APTCA behavior is different depending on the version of the framework, but the general effect is the same: potentially dangerous functionality is now exposed to partially trusted code, and the effects of this additional attack surface exposure have to be mitigated by security checks, strong enforcement mechanisms, and review. In order to comply with this requirement, you should follow these steps:

  1. Determine whether or not your assembly has enabled the AllowPartiallyTrustedCallersAttribute. You may do this by either looking for the annotations in your source code or using tools like FxCop which will flag usage. If you determine that this attributes is not enabled, you are compliant with this requirement, no review is required, and you may skip the rest of these steps.
  2. If your assembly has enabled the AllowPartiallyTrustedCallers attribute, your assembly needs to be reviewed by project security experts.
- For online services, all new releases must use the Relying Party Suite (RPS) v4.0 SDK. RPS provides significant security advantages over the current Passport Manager (PPM) SDK; most important being the elimination of the shared symmetric encryption keys, which mitigates security issues involving key distribution, deployment, and administration. This also provides a significantly reduced cost of key revision.
- **(Updated for SDL 5.2)** User Account Control (UAC) is a security feature in Windows Vista, Windows 7, Windows Server® 2008, and Windows Server 2008 R2. It is intended to help the transition to regular use of non-administrative privilege by client applications.

- **(Updated for SDL 5.2)** Comply with UAC best practices and ensure that your application runs with least privilege whenever possible. Exit criteria for this requirement is confirmation from the project team that it has analyzed and minimized the need for elevated privileges and followed best practices for operation in a UAC environment. Highly privileged features of the application (such as those that perform administrative functions) are separated out and placed behind an elevation prompt as appropriate. Following this requirement enables teams to design and develop their applications with a standard user in mind. This results in a reduced attack surface exposed by applications, which increases the security of the user and the system.
- If a program's users require an open port in the firewall, then the code that listens on the port must comply with certain quality requirements. Prohibited and permitted actions are covered in the bulleted list that follows. See Appendix D: Firewall Rules and Requirements for additional information. For a discussion of Windows Firewall integration and best practices, please visit http://msdn.microsoft.com/en-us/library/bb736286(VS.85).aspx.

  The following actions are prohibited:

  - Except for security products, disabling of the firewall or changing the state of the firewall. The firewall must only be disabled by explicit user action.
  - Any service or feature that adds, changes, or removes firewall rules automatically at runtime. Except at setup time (that is, during the installation process), programs, features, and services that are not designed specifically as firewall management utilities must not change firewall settings unless the user has explicitly initiated some action.
  - Any service or feature that allows a port to be opened or a rule to be enabled by a user without administrative privileges. A user must be acting as an administrator in order to change the settings of the firewall, and no service or feature (both Windows and non-Windows) must bypass this restriction.
  - Silent activation or enabling of any feature that permits other programs to receive unsolicited traffic. For example, the RemoteAdmin feature permits other RPC-based programs to receive unsolicited traffic. In such cases, the system must obtain user consent before activating such functionality.
  - Programs, services, and features may not configure an external device (for example, a NAT gateway) without user consent.
  - Any interference with Post Setup Security Update or similar functionality designed to ensure that the system is up-to-date prior to accepting incoming traffic without user consent.
  - Creation of inbound firewall rules unless the feature or service will receive unsolicited inbound traffic.

The following actions are permitted:

- Programs, services, and features may define a firewall rule and leave it disabled for the sake of making it convenient for the user to enable the rule later on.

- All cryptography must comply with the Microsoft Cryptographic Standards for SDL-covered products. Adhere to the SDL crypto requirements, which at a high-level are:
  - Use AES for symmetric enc/dec.
  - Use 128-bit or better symmetric keys.
  - Use RSA for asymmetric enc/dec and signatures.
  - Use 2048-bit or better RSA keys.
  - Use SHA-256 or better for hashing and message-authentication codes.
  - Support certificate revocation.
  - Limit lifetimes for symmetric keys and asymmetric keys without associated certificates.
  - Support cryptographically secure versions of SSL (must not support SSL v2).
  - Use cryptographic certificates reasonably and choose reasonable certificate validity periods.
  - **(New for SDL 5.2) Use Transport Layer encryption securely.** Properly use Transport Layer Security (TLS) when communicating with another entity, and verify that your service checks the **Common Name** attribute to be sure it matches the host with which you intended to communicate. Verify that your service consults a certificate revocation list (CRL) for an updated list of revoked certificates at a frequent interval. If your service is accessible via a browser, confirm that no security warnings appear at any visited URL for any supported browser.
- Mitigate against Cross-Site Scripting (XSS), which is a client-side code injection attack that allows arbitrary code execution in your customer's browser. XSS has been used by attackers to capture credentials, financial data, and other sensitive information. It has also been used to map back-end server space and in cases of vulnerable browser plug-ins, completely compromise a customer's machine. Perform all pre-approved tests that have been explicitly approved by your security advisor. If this is an ASP.NET application, this should include use of CAT.NET. All test results should be uploaded into your project tracking system to validate against project requirements. All issues found by approved tests must be triaged with the security advisor and fixed in accordance with the SDL bug bar. Suggested tools include: Web Protection (Anti-XSS) Library, CAT.NET, and Watcher (from Casaba Security for the MAC portion).

Privacy Requirements

- If your project has a Privacy Impact Rating of P1, identify a compliant design based on the concepts, scenarios, and rules in the Microsoft Privacy Guidelines for Developing Software Products and Services. Definitions of privacy rankings (P1, P2, P3) can be found in Cost Analysis and Chapter 8 of

The Security Development Lifecycle. You can find additional guidance in Appendix C: SDL Privacy Questionnaire.

Security Recommendations

- Include in all functional and design specifications a section that describes impacts on security.
- Write a security architecture document that provides a description of a software project that focuses on security. Such a document should complement and reference existing traditional development collateral without replacing it. A security architecture document should contain, at a minimum:
  - **(Updated for SDL 5.2)** Attack surface measurement. After all design specifications are complete, define and document what the program's default and maximum attack surfaces are. The size of the attack surface indicates the likelihood of a successful attack. Therefore, your goal should be to minimize the attack surface. You can find additional background information in the papers Fending Off Future Attacks by Reducing Attack Surface and Measuring Relative Attack Surfaces. Use of tools like Attack Surface Analyzer (ASA) and Web Application Configuration Analyzer (WACA) should be required for existing products to identify the existing attack surface.
  - Product structure or layering. Highly structured software with well-defined dependencies among components is less likely to be vulnerable than software with less structure. Ideally, software should be structured in a layered hierarchy so that higher components (layers) depend on lower ones. Lower layers should never depend on higher ones. Developing this sort of layered design is difficult and might not be feasible with legacy or pre-existing software. However, teams that develop new software should consider layered and highly structured designs.
- Minimize default attack surface/enable least privilege.
  - All feature specifications should consider whether the features should be enabled by default. If a feature is not used frequently, you should disable it. Consider carefully whether to enable by default those features that are used infrequently.
  - If the program needs to create new user accounts, ensure that they have as few permissions as possible for the required function and that they also have strong passwords.
  - Be very aware of access control issues. Always run code with the fewest possible permissions. When code fails, find out why it failed and fix the problem instead of increasing permissions. The more permissions any code has, the greater its exposure to abuse.
- Default installation should be secure. Review functionality and exposed features that are enabled by default and constitute the attack surface carefully for vulnerabilities.
- Consider a defense-in-depth approach. The most exposed entry points should have multiple protection mechanisms to reduce the likelihood of exploitation of any security vulnerabilities that might exist. If possible, review public sources of information for known vulnerabilities in competitive products, analyze them, and adjust your product's design accordingly.

- If the program is a new release of an existing product, examine past vulnerabilities in previous versions of the product and analyze their root causes. This analysis might uncover additional instances of the same classes of problems.
- Deprecate outdated functionality. If the product is a new release of an existing product, evaluate support for older protocols, file formats, and standards, and strongly consider removing them in the new release. Older code written when security awareness was less prevalent almost always contains security vulnerabilities.
- Conduct a security review of all sample source code released with the product and use the same level of scrutiny as for object code released with the product.
- If the product is a new release of an existing product, consider migration of any possible legacy code from unmanaged code to managed code.
- Implement any new code using managed code whenever possible.
- When developing with managed code, take advantage of .NET security features:
  - Refuse unneeded permissions.
  - Request optional permissions.
  - Use CodeAccessPermission Assert and LinkDemand carefully. Use Assert in as small a window as possible.
  - Disable tracing and debugging before deploying ASP.NET applications.
- Watch for ambiguous representation issues. Hackers will try to force code to follow a dangerous path or URL by hiding their intent in escape characters or obscure conventions. Always design code to deal with full canonical representations, rather than acting on externally provided data. The canonical representation of something is the standard, most direct, and least ambiguous way to represent it.
- Remain informed about security issues in the industry. Attacks and threats evolve constantly, and staying current is important. Keep your team informed about new threats and vulnerabilities.
- Ensure that everyone on your team knows about unsafe functions and coding patterns. Maintain a list of your code's vulnerabilities. When you find new vulnerabilities, publish them. Make security everyone's business.
- Be careful with error messages. Sensitive information displayed in an error message can provide an attacker with privileged information, such as a file path on a server or the structure of a query. Such information makes it easier for an attacker to attack any defenses. In general, record detailed failure messages in a secure log, and give the user discreet failure messages.
- For online services and/or LOB applications, ensure appropriate logging is enabled for forensics.
- For online services and/or LOB applications, page flow integrity checking should be performed.

- Hardware security design review. Conduct a high-level security design review for hardware products that are new or being updated in the current release. The goal of a high-level hardware security design review is to identify aspects of the design that could lead to security vulnerabilities. This might include checks such as:
  1. Methods of cryptographic key generation and storage.
  2. Methods of data storage, including encryption, For example, does the design meet Microsoft Cryptographic Standards?
  3. Methods of data manipulation.
  4. The "business or customer impact" of the data being manipulated and stored. For example, whether the data could be considered High Business Impact (HBI), Moderate Business Impact (MBI), or Low Business Impact (LBI) by the product's expected customers.
  5. Use of standard versus custom protocols.
  6. Presence of JTAG*/debugging back doors.

     * Joint Test Action Group (JTAG) is the common name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for testing/debugging access ports using a boundary scan.
  7. Firmware upgrade features and procedures (security, process, and scalability).
  8. Firmware development process – /analyze, fuzzing and/or other security tools may be applicable.
- Integration-points security design review. Enterprise server application suites, software-as-a-service (SaaS) offerings, and security products interact with a variety of other products and platforms in order to provide robust, enterprise-focused services. It is not sufficient to threat model these products by themselves because the points of interaction with other products are the source of many nuanced security issues. Conduct an integration-points security design review with dependent product teams across your end-to-end scenarios. Examples of products that should do this are:
  - Products designed to handle high business impact (HBI) data.
  - Enterprise applications and services.
  - Security products.
  - SaaS offerings.

  The exit criteria for this recommendation is that the product teams and security reviewers/owners have reviewed and are satisfied with the security threat mitigations and validations provided at each point of integration.
- Strong log-out and session management. Proper session handling is one of the most important parts of web application security. At the most fundamental level, sessions must be initiated, managed, and terminated in a secure manner. If a product employs an authenticated session, it must begin as an encrypted authentication event to avoid session fixation.

- A session identifier/token must never be transmitted via the URL to avoid side-jacking via the referrer header or browser history.
- All session data must be maintained on the server, not the client, to avoid tampering.
- Sessions must be completely terminated on the server side via logout and timeout mechanisms. When multiple sessions are tied to a single authentication event, all of the sessions tied to that event must be terminated by logout/time-out.

The following items must be met as part of the exit criteria for this recommendation:

- When multiple sessions are tied to a single user identity, they must be collectively terminated on the server side at timeout or logout.
- Authentication events must invalidate unauthenticated sessions and create a new session identifier.
- Logout functionality is available on every page.
- Session state, outside of a single identifier, is maintained on the server and not accepted from the user (including via cookie or header).
- Session tokens are not present in the URI.
- Timeout functionality is present and timeout thresholds are documented along with the rationale.

- Apply no-open header to user-supplied downloadable files. Use the HTTP Header X-Download-Options: noopen for each HTTP file download response that may contain user-controllable content. Recommended tool: Casaba Passive Security Auditor.

Privacy Recommendations

- If your project has a privacy impact rating of P2, identify a compliant design based on the concepts, scenarios, and rules in the Microsoft Privacy Guidelines for Developing Software Products and Services. Additional guidance can be found in Appendix C: SDL Privacy Questionnaire.
- Use FxCop to enforce design guidelines in managed code. Many rules are built in by default.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 8: Stage 3—Product Risk Assessment
- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 20: SDL Minimum Cryptographic Standards (pp. 251-258)
- *Writing Secure Code, Second Edition* (ISBN 9780735617223; ISBN-10 0-7356-1722-8), Appendix C: A Designer's Security Checklist (p. 729)
- Fending Off Future Attacks by Reducing Attack Surface, an MSDN article on the process for determining attack surface
- Measuring Relative Attack Surfaces, a more in-depth research paper

Risk Analysis

During the design phase of development, carefully review security and privacy requirements and expectations to identify security concerns and privacy risks. It is efficient to identify and address these concerns and risks during the Design phase.

For security concerns, *threat modeling* is a systematic process that is used to identify threats and vulnerabilities in software. You must complete threat modeling during project design. A team cannot build secure software unless it understands the assets the project is trying to protect, the threats and vulnerabilities introduced by the project, and details of how the project mitigates those threats. Threat modeling applies to all products and services, all code types, and all platforms. Verify that threat models exist for all attack surfaces and new features and each threat model includes: diagrams showing the software and all trust boundaries, STRIDE threats enumerated for each element that crosses a trust boundary or that connects to a data flow that crosses a trust boundary, mitigations for all threats, a list of assumptions made while threat modeling, all non-platform external dependencies that the elements in the threat model rely on.

Important risk analysis considerations include the following:

- Threats and vulnerabilities that exist in the project's environment or that result from interaction with other systems. You cannot consider the design phase complete unless you have a threat model or models that include such considerations. Threat models are critical components of the design phase and reference a project's functional and design specifications to describe vulnerabilities and mitigations.
- Code that was created by external development groups in either source or object form. It is very important to carefully evaluate any code from sources external to your team. Failure to do so might cause security vulnerabilities about which the project team is unaware.
- Threat models that include all legacy code, if the project is a new release of an existing program. Such code could have been written before much was known about software security and therefore could contain vulnerabilities.
- A review of the design of high-risk (P1) privacy projects with a privacy subject-matter expert (SME) and, if necessary, with appropriate legal counsel conducted as soon as possible in the project. Definitions of privacy rankings (P1, P2, P3) can be found in Cost Analysis and Chapter 8 of *The Security Development Lifecycle*.
- A detailed privacy analysis to document your project's key privacy aspects. Important issues to consider include:
  - What personal data is collected?
  - What is the compelling user value proposition and business justification?

- What notice and consent experiences are provided?
- What controls are provided to users and enterprises?
- How is unauthorized access to personal information prevented?

Security Requirements

- Complete threat models for all functionality identified during the cost analysis phase. Threat models typically must consider the following areas:
  - All projects. All code exposed on the attack surface and all code written by or licensed from a third party.
  - New projects. All features and functionality.
  - Updated versions of existing projects. New features or functionality added in the updated version.
- Ensure that all threat models meet minimal threat model quality requirements. All threat models must contain data flow diagrams, assets, vulnerabilities, and mitigation. Threat modeling can be done in a variety of ways, using either tools or documentation/specifications to define the approach. For assistance in creating threat models, see "Chapter 9: Stage 4 – Risk Analysis" in *The Security Development Lifecycle* book or consult other guidance listed in Resources.
- Have all threat models and referenced mitigations reviewed and approved by at least one developer, one tester, and one program manager. Ask architects, developers, testers, program managers, and others who understand the software to contribute to threat models and to review them. Solicit broad input and reviews to ensure the threat models are as comprehensive as possible.
- Confirm that threat model data and associated documentation (functional/design specifications) has been stored using the document control system used by the product team.

Privacy Requirements

If a project has a privacy impact rating of P1:

- Complete Detailed Privacy Analysis in Appendix C: SDL Privacy Questionnaire. The questions will be customized to the behaviors specified in the initial assessment.
- Hold a design review with your privacy subject-matter expert.

If your project has a privacy impact rating of P2:

- Complete the Detailed Privacy Analysis in Appendix C: SDL Privacy Questionnaire. The questions will be customized to the behaviors specified in the initial assessment.
- Hold a design review with your privacy subject-matter expert only if one or more of these criteria apply:
  - The privacy subject-matter expert requests a design review.
  - You want confirmation that the design is compliant.

- You wish to request an exception.

If your project has a privacy impact rating of P3, there are no privacy requirements during this phase.

Security Recommendations

- The person who manages the threat modeling process should complete threat modeling training before working on threat models.
- After all specifications and threat models have been completed and approved, the process for making changes to functional or design specifications—known as design change requests (DCRs—should include an assessment of whether the changes alter existing threats, vulnerabilities, or the effectiveness of mitigations.
- Create an individual work item for each vulnerability listed in the threat model so that your quality assurance team can verify that the mitigation is implemented and functions as designed.
- **(New for SDL 5.2)** Follow NEAT security user experience (UX) guidance to ensure that warnings are **N**ecessary, **E**xplained, **A**ctionable, and **T**ested.
  - Microsoft has produced guidance to help engineers design better warnings. The guidance is summarized in the four-letter acronym NEAT—Necessary, Explained, Actionable, and Tested. The objective of NEAT is to improve security warnings in products by making sure they are:
    - **Necessary.** Does the user really need to be presented with the decision?
    - **Explained.** Does the UX present all the information the user needs to make this decision?
    - **Actionable.** Is there a set of steps users can take to make good decisions in both benign and malicious scenarios?
    - **Tested.** Has the warning been reviewed by multiple engineers to make sure users will understand how to respond to the warning?
  - Improve security-related prompts in your products to enable customers to make good security decisions that protect them from harm.
  - Users face a barrage of decisions about whom and what to trust, and when to be concerned about their security. These security decisions arise when users initiate activities like installing an executable from the web, using an application that needs to get through the firewall, and allowing a web application to access their sensitive data. These decisions also arise when a product prompts the user to take an action to improve their security, such as when Windows prompts the user to reboot or to turn the Windows Firewall on. With attacks moving "up the stack" from silent code exploits to social engineering, good security UX gives you a chance to help the user avoid running malicious software.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 8: Stage 3—Product Risk Assessment
- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 9: Stage 4—Risk Analysis
- Threat Modeling: Uncover Security Design Flaws Using The STRIDE Approach
- SDL Threat Modeling Tool
- Fending Off Future Attacks by Reducing Attack Surface, an MSDN article that describes the process for determining attack surface
- Measuring Relative Attack Surfaces, a more in-depth research paper

Phase Three: Implementation

The Implementation phase is when the end user of your software is foremost in your mind. During this phase you create the documentation and tools the customer uses to make informed decisions about how to deploy your software securely. To this end, the Implementation phase is when you establish development best practices to detect and remove security and privacy issues early in the development cycle.

Creating Documentation and Tools for Users That Address Security and Privacy

Every release of a software program should be secure by design, in its default configuration, and in deployment. However, people use programs differently, and not everyone uses a program in its default configuration. You need to provide users with enough security information so they can make informed decisions about how to deploy a program securely. Because security and usability might conflict, you also need to educate users about the threats that exist and the balance between risk and functionality when deciding how to deploy and operate software programs.

It is difficult to discuss specific security documentation needs before development plans and functional specifications stabilize. As soon as the architecture is reasonably stable, the user experience (UX) team can develop a security documentation plan and schedule. Delivering documentation about how to use a software program securely is just as important as delivering the program itself.

Security Recommendations

- Development management, program management, and UX teams should meet to identify and discuss what information users will need to use the software program securely. Define realistic use and deployment scenarios in functional and design specifications. Consider user needs for documentation and tools.
- User experience teams should establish a plan to create user-facing security documentation. This plan should include appropriate schedules and staffing needs. Communicating the security aspects of a program to the user in a clear and concise fashion is as important as ensuring that the product code or functionality is free of vulnerabilities.
- For new versions of existing programs, solicit or gather comments about what problems and challenges users faced when securing prior versions.
- Make information about secure configurations available separately or as part of the default product documentation and/or help files. Consider the following issues:
    - The program will follow the best practice of reducing the default attack surface. However, what should users know if they need to activate additional functionality? What risks will they be exposed to?
    - Are there usage scenarios that allow users to lock down or *harden* the program more securely than the default configuration without losing functionality? Inform users about how to configure the program for these situations. Better yet, provide easy-to-use templates that implement such configurations.
    - Inform users about security best practices, such as removing guest accounts and default passwords. External security notes from threat modeling are good sources of information to consider.
    - For programs that use network or Internet communications, describe all communications channels and ports, protocols, and communications configuration options (and their associated security impacts).
    - To support earlier versions of the software and older protocols, it is often necessary to operate less securely. Do not enable insecure protocols in the default configuration. You might still need to deliver them with the release, so inform users about the security implications of older protocols and backward compatibility. Inform users about these trade-offs and how to disable older compatibility modes to achieve the best possible security.
    - Tell users how they can ensure safe use of the program or take advantage of built-in security and privacy features.

Privacy Recommendations

- If the program contains privacy controls, create deployment guides for organizations to help them protect their users' privacy (for example, **Group Policy** controls).
- Create content to help users protect their privacy when using the program (for example, secure your subnet).

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 10: Stage 5—Creating Security Documents, Tools, and Best Practices for Customers
- Templates contained in the *Windows Server 2003 Security Guide*

Establish and Follow Best Practices for Development

To detect and remove security issues early in the development cycle, it helps to establish, communicate, and follow effective practices for developing secure code. A number of resources, tools, and processes are available to help you accomplish this goal. Investing time and effort to apply effective practices early will help you eliminate security issues and avoid having to respond to them later in the development cycle—or even after release, which is expensive.

Security Requirements

- **(Promoted for SDL 5.2)** Components must have no hard dependencies on the NTLM protocol. All explicit uses of the NTLM package for network authentication must be replaced with the Negotiate package. All client authentication calls must provide a properly formatted target name—service principal name (SPN). The purpose of the requirement is to enable systems to use Kerberos in place of NTLM whenever possible.
- **(Promoted for SDL 5.2) HTTPOnly Cookies.** To mitigate the risk of information disclosure with a cross-site scripting attack, a new attribute was introduced to cookies for Internet Explorer 6 Service Pack 1 (SP1) and is now in use in all current browsers. This attribute specifies that a cookie is not accessible through script. By using HTTP-only cookies, a web application reduces the possibility that sensitive information contained in the cookie can be stolen via script. Watcher, a web security testing tool, can help you meet this recommendation.

  All HTTP-based applications that use cookies must specify HttpOnly in the cookie definition for all cookies not explicitly required by legitimate scripts in the web page. For example:

```
Set-Cookie: USER=123; expires=Wednesday, 10-Feb-2012 23:12:40 GMT;
HttpOnly"
```

- Use minimum code generation suite and libraries. For unmanaged, native C/C++ code, use Visual C++ 2010 as it offers all the SDL-mandated compiler and linker flags, including /GS, /DYNAMICBASE, /NXCOMPAT, and /SAFESEH. For managed code, use Visual Studio® 2008 SP1 or later. Use the currently required (or later) versions of compilers to compile options for the Win32®, Win64, WinCE, and Macintosh target platforms, as listed in Appendix E: SDL Required and Recommended Compilers, Tools, and Options for All Platforms. The biggest change in Visual Studio 2008 SP1 and later is Data Execution Prevention (DEP) support, enabled by default for all binaries, which can help protect against classes of buffer overrun.

  For unmanaged C or C++ code, BinScope must indicate a "Pass" in the compiler version field for all binaries. For managed code, an attestation is required that the compiler version used to ship the product is the version outlined in this document or later.
- Code analysis tools. Use the currently required (or later) versions of code analysis tools for either native C and C++ or managed (C#) code that are available for the target platforms, as listed in Appendix E: Required and Recommended Compilers, Tools, and Options for All Platforms. Run PREfast for Drivers on all kernel-mode driver source code. File bugs for each instance of each warning as required by the SDL and triage based on tool output. Ensure that all appropriate bugs are fixed.
- Banned application programming interfaces (APIs). All native C and C++ code must not use banned versions of string buffer handling functions. Based on analysis of previous Microsoft Security Response Center (MSRC) cases, avoiding use of banned APIs is one actionable way to remove many vulnerabilities. For more information, see Security Development Lifecycle (SDL) Banned Function Calls and *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 19: SDL Banned Function Calls (pp. 241-49).
- No writable shared PE sections. Sections marked as *shared* in shipping binaries represent a security threat. Use properly secured dynamically created shared memory objects instead. See Appendix G: SDL Requirement: No Shared Sections.
- For online services and/or LOB applications, follow data input validation and output encoding requirements to address potential cross-site scripting vulnerabilities.
- For online services and/or LOB applications that access a SQL database, do not use “ad-hoc” SQL queries in order to avoid SQL injection attacks.
- For online services and/or LOB applications that implement web services, use an approved XML parser.
- Fix issues identified by code analysis tools for managed code. Run the FxCop code analysis tool against all managed code, and fix all violations of the “Security” rules for the version of FXCop used. Note that there are slight differences in the security rules for FXCop 1.35 (from Visual Studio 2005) and FXCop 1.36 (from Visual Studio 2008). These differences are explained in the Code Analysis Team Blog. When using .NET Framework version 3.5, FXCop 1.36 must be used. Security rules for FXCop can

be found at http://msdn.microsoft.com/en-us/library/ms182296(VS.80).aspx. Security rules for FXCop can be found at http://www.microsoft.com/en-us/download/details.aspx?id=6544.

- Compile native code with /GS compiler. All unmanaged C and C++ code must be compiled with the /GS compiler option. /GS- (which turns off /GS) is not allowed. Verify all build files include the /GS compiler option so that all native code C and C++ binaries are compiled using this option. High-risk code (code facing the Internet, file parsers, and ActiveX controls) must have #pragma strict_gs_check(on) in an application-wide header file, such as stdafx.h. Ensure no buffers are marked as safebuffers. Verify there are no instances of compiler warning C4748.
- Address Space Layout Randomization (ASLR) must be enabled on all native code (unmanaged) binaries to protect against return-to-libc class of attacks. Enabling this functionality requires the flag /DynamicBase in the PE header of all binaries. This flag can be inserted using Visual Studio 2005 SP1 or later. Earlier versions do not contain a linker version that supports it. DumpBin can be used to manually verify if ASLR is enabled on a binary.
- Do not use banned APIs. Problems arise when an attacker controls the incoming buffer and the code uses data from the buffer to determine the maximum buffer length to copy. Verify there are no banned APIs in shipping code, including sample code. Recommended tool: banned.h header or use of Visual C++ use of C4996 warnings.
- Use secure methods to access databases. Creating dynamic queries using string concatenation potentially allows an attacker to execute an arbitrary query through the application. Verify that all database access is performed through a combination of parameterized inline queries, stored procedures (activated through parameterized queries or LINQ), or object names passed to Windows Azure Storage APIs that are not based on user-provided data.

  Also verify:

  - If directly calling the Windows Azure Storage Blob REST APIs, all parameters should be URL-encoded with AntiXSS.Urlencode. When calling Windows Azure Table APIs, WCF Data Services should be used rather than writing REST queries directly.
  - The database is accessed via a least-privilege account with only the minimum level of access required to carry out the application's functions. This account must never have system administrator (SA), database owner (DBO), or other administrative privileges.
- Avoid LINQ ExecuteQuery. LINQ queries are normally converted by the compiler/framework into parameterized queries to be passed to the database. However, the LINQ method ExecuteQuery allows the developer to pass arbitrary SQL commands that may have been created through string concatenation. Creating dynamic queries using string concatenation potentially allows an attacker to execute an arbitrary query through the application. This vulnerability allows for unauthorized, interactive logon to a SQL server that may result in the execution of malicious commands leading to the possible disclosure, modification, or deletion of the operating system or user data.

Calls to System.Data.Linq.DataContext.ExecuteQuery are prohibited. If you are using LINQ, use the standard LINQ object-relational mapping (ORM) syntax or stored procedure syntax.

- Examples of correct code/procedures

```
Correct ORM:
from product in Products select product

Correct stored procedure:
SelectProducts()
```

- Example of incorrect code:

```
ExecuteQuery<Products>("SELECT * FROM Products")
```

The exit criteria for this requirement is that your code contains no references to System.Data.Linq.DataContext.ExecuteQuery.

- Avoid EXEC in stored procedures. Verify that the stored procedures contain no calls to EXEC, EXECUTE, or sp_executesql (except for calls to other stored procedures).
- Do not use Microsoft Visual Basic® 6 to build products. Make sure that all Visual Basic code in the product, including sample code, is built with Visual Basic .NET and not Visual Basic 6. No product distributes or uses the runtime environment for Visual Basic 6. When running BinScope, confirm it reports no instances of binaries built with Visual Basic 6.
- Harden or disable XML entity resolution. XML parsing code can be vulnerable to exponential entity expansion attacks and external entity resolution attacks causing denials of service and disclosure of sensitive data. If XML entity resolution is not required by your application, then disable it. Verify that all XML parsing code (including use of XmlReader and XmlDocument) do one of the following:
  - Completely disable entity resolution.
  - Limit the size of internal entity resolution and disable external entity resolution if it is not possible to disable entity resolution.
  - Limit the size of internal entity resolution and limit the time and size of external entity resolution if it is not possible to disable external entity resolution.
- Use safe integer arithmetic for memory allocation for new code. All new code that uses arithmetic to determine the amount of dynamic memory to allocate must be safe from any form of overflow, underflow, or truncation.
- Use secure cookie over HTTPS. HTTP cookies created over HTTPS should not be visible to the same site over clear text via HTTP. Ensure that all cookies set over HTTPS use the "secure" attribute. Suggested tools include: Watcher (from Casaba Security).

- AllowPartiallyTrustedCallersAttribute (APTCA) review. Ensure that the AllowPartiallyTrustedCallersAttribute (APTCA) enables an assembly to be called by untrusted code. APTCA behavior is different depending on the version of the framework, but the general effect is the same: potentially dangerous functionality is now exposed to partially trusted code, and the effects of this additional attack surface exposure have to be mitigated by security checks, strong enforcement mechanisms, and review. Ensure there is no code using APTCA or that your project exception review process has approved a request to use APTCA.
- Mitigate against cross-site request forgery (CSRF). Cross-site request forgery is a type of attack in which a malicious website exploits the user's browser to send commands to another website. This attack exploits the victim site's trust in the user's browser, generally by using the user's cookies or session.
    - For ASP.NET or ASP.NET MVC projects, verify that all pages in the application report no AlwaysSetViewStateUserKeyTokenValue or MarkVerbHandlersWithValidateAntiforgeryToken FxCop rule violations.
    - For non ASP.NET projects, verify that:
        - A secondary token is submitted with each request. Ideally, the token must be unique per user, though session-unique tokens will suffice.
        - Validation tokens are not automatically submitted (such as via cookie). Including a hidden input value in a POST is the preferred method.
        - The token is not predictable. If the attacker can predict the token, it negates the protection.
- Load DLLs securely. Applications and dynamic-link libraries (DLLs) must load DLLs securely to prevent DLL preloading vulnerabilities.
- Minimum ATL Version and Secure COM Coding Requirements. Developers using the Active Template Library (ATL) to build COM controls must make sure they use the most secure ATL versions and use appropriate secure ATL coding constructs. Verify that the latest version of the ATL library is used and all ATL code is implemented with the correct secure coding constructs. This can be verified by passing the ATLVersionCheck and ATLVulnCheck tests with BinScope.
- Reflection and authentication relay defense. This new recommendation is designed to help combat sophisticated toolkits that are available to implement reflection and relay attacks. It is hoped that it will aid developers in utilizing available defenses against reflection and authentication relay attacks. For network authentication, the following tasks should be followed:
    1. Specify the target system name as part of the authentication mechanism.
    2. Ensure channel integrity through signing and/or encrypting underlying transport messages using the negotiated session key, or implement Extended Protection for Authentication when signing or encryption is not supported.

- Sample code should be SDL compliant. All sample code (code snippets, small sample applications, or complete sample applications) should meet the same SDL development practices security bar as if it were in a shipping product. That means all the appropriate tools must be run, and there is no use of banned functionality or compiler switches.
- Internet Explorer 8 MIME handling: Sniffing OPT-OUT. This recommendation addresses functionality new in Internet Explorer 8 that may have security implications in some cases. It is recommended that for each HTTP response that could contain user controllable content, you utilize the HTTP Header X-Content-Type-Options:nosniff. The Watcher tool may be of use in meeting this requirement.
- Safe redirect, online only. Automatically redirecting the user (through Response.Redirect, for example) to any arbitrary location specified in the request (such as a query string parameter) could open the user to phishing attacks. Therefore, it is recommended that you not allow HTTP redirects to arbitrary user-defined domains.
- Comply with minimal Standard Annotation Language (SAL) code annotation recommendations as described in Appendix H: SDL Standard Annotation Language (SAL) Recommendations for Native Win32 Code. Annotating code helps existing code analysis tools identify implementation issues better and also helps improve the tools. SAL annotated code has additional code analysis requirements, as described in SDL SAL Recommendations.

Privacy Requirements

Establish and document development best practices for the development team. Communicate any design changes that affect privacy to your team's privacy lead so that they can document and review any changes.

Security Recommendations

- Use HeapSetInformation. Use Windows heap corruption detection to help reduce the likelihood of successful exploitation of residual heap-based vulnerabilities.
- Review available information resources to adopt appropriate coding techniques and methodologies. For a current and complete list of all development best practice information and resources, see *Writing Secure Code, Second Edition* (ISBN 9780735617223; ISBN-10 0-7356-1722-8).
- Review recommended development tools and adopt appropriate tools.
- Define, document, and communicate to your entire team all best practices and policies based on analysis of all the resources and tools listed in this document.
    - Document all tools that are used, including compiler versions, compile options (for example, /GS), and additional tools used. Also, forecast any anticipated changes in tools. For more information about minimum tool requirements and related policy, review How Are New Recommendations and New Requirements Added to the Security Development Life Cycle Process?

- Create a coding checklist that describes the minimal requirements for any checked-in code. This checklist can include some of the items from *Writing Secure Code, Second Edition* "Appendix D: A Developer's Security Checklist" (p. 731), clean compile warning level requirements (/W3 as minimal and /W4 clean as ideal), or other desired minimum standards.
- Establish and document how the team enforces these practices. Is the team running scripts to check for compliance when code is checked in? How often do you run analysis tools? The development manager is ultimately responsible for establishing, documenting, and validating compliance of development best practices.
- For online services and/or LOB applications that use JavaScript, avoid use of the **eval()** function.
- Additional development best practices for security can be divided into three general categories:
  1. Review available information resources to adopt coding techniques and methodologies that are appropriate for the product.
  2. Review recommended development tools to adopt, and use those that are appropriate for the product, in addition to the tools required by the SDL.
  3. Define, communicate, and document all best practices and policies for the product. Based on analysis of all of the resources and tools listed above, product teams should adopt and communicate best practices, policies, and tools. This information should be documented and widely communicated to the entire product team to ensure best practices are adopted and followed.
- Document all tools used. This includes all compiler versions, compile options (for example, /GS), and additional tools used for static code analysis. This should also forecast any changes in tools anticipated. As updated versions of tools (both compilers and code analysis tools) are made available, products that have not yet released a final beta will likely be required to adopt the newest version of the tools. Products that have released their final beta prior to the availability of updated tools are not required to adopt the latest versions of tools. Please review How Are New Recommendations and New Requirements Added to the Security Development Lifecycle Process? For more information on how policy is established on minimum tool requirements.
- Use the Windows Imaging Component (WIC). WIC provides an extensible framework for reading and manipulating images, image files, and image metadata. It represents a standard interface. All software products that process digital image data must perform any encoding or decoding of image data solely and exclusively using the Windows Imaging Component (WIC) and therefore must remove any potential custom (image) codecs from the product codebase. For more information, see http://msdn.microsoft.com/en-us/library/ee719654.aspx.
- Ensure that regular expressions must not execute in exponential time (O(2^n)). Regular expressions that are evaluated against untrusted input must be examined for unsafe patterns that can lead to denial-of-service (DoS) vulnerabilities. Use RegexFuzzer to analyze all regular

expressions to ensure that they are free of grouping expressions with repetition that are themselves repeated and containing alternation where the alternate sub-expressions overlap each other.

- Use proper http.sys URL canonicalization. Web servers often make security decisions based on a requested URL from a user. For example, a web server may deny access to certain files, such as configuration files, directly through the web server; rather, such files can only be viewed and manipulated with text editing tools, such as Visual Studio, against the local file system.

  The weakness with this approach is that the authorization check is based on the name of the resource, rather than based on an access control mechanism, such as an ACL, enforced by the operating system. And this leads to canonicalization (C14N) vulnerabilities, because there is often more than one way to name a resource. All web servers have suffered from such issues, but http.sys does not offer much protection from C14N vulnerabilities. It is therefore important that developers using http.sys follow these recommendations to defend themselves. Any application that uses http.sys should follow these guidelines.

  Managed Code

  1. Limit the URL length to no more than 16,384 characters (ASCII or Unicode). This is the absolute maximum URL length, based on the default IIS 6 setting. websites should strive for a length shorter than this, if possible.
  2. Use the standard .NET Framework file I/O classes (such as FileStream), since these take advantage of the canonicalization rules in the .NET FX.
  3. Explicitly build an allow-list of known filenames.
  4. Explicitly reject known filetypes you will not serve; UrlScan rejects: exe, bat, cmd, com, htw, ida, idq, htr, idc, shtm[l], stm, printer, ini, pol, dat files.
  5. Catch the following exceptions: System.ArgumentException (for device names), System.NotSupportedException (for data streams), System.IO.FileNotFoundException (for invalid escaped filenames), and System.IO.DirectoryNotFoundException (for invalid escaped dirs).
  6. Do *not* call out to Win32 file I/O APIs.
  7. On an invalid URL, gracefully return a 400 error to the user, and log the real error.

  Unmanaged Code

  1. Limit the URL length to 16,384 characters (ASCII or Unicode).
  2. Prepend \\?\ to the filename prior to accessing the file system, since this forces the file system to bypass filename equivalency checks. This is not perfect, because it does not prevent data streams (::$DATA, for example).
  3. Normalize the URL by URL double-decoding the filename, and check that the first decode matches the second decode. If not, it's an error.

4. Look for known “bad characters” in the filename.
5. Look for known “bad strings” in the filename.
6. On an invalid URL or filename, return a 400, and log the real error.

- Use Standard Annotation Language (SAL). Annotate all functions that read from or write to a buffer passed as an argument to the function.
- Do not use the JavaScript **eval()** function (or equivalents). The JavaScript **eval()** function is used to interpret a string as executable code. While **eval()** enables a web application to dynamically generate and execute JavaScript (including JSON), it also opens up potential security holes, such as injection attacks, where an attacker-fed string may also get executed. For this reason, the **eval()** function or functional equivalents, such as **setTimeout()** and **setInterval()**, should not be used.
- Enforced automated banned API replacement. Add the following to an often-used header file, such as stdafx.h:

  ```
  #define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES (1)
  ```

  This informs the compiler that you want to upgrade various C runtime functions to safer versions. For example, some calls to strcpy will upgrade to strcpy_s. For more information on banned APIs, visit http://blogs.msdn.com/sdl/archive/2008/10/22/good-hygiene-and-banned-apis.aspx.
- Encode long-lived pointers. Long-lived pointers (for example, globally scoped function pointers or pointers to shared memory regions) are subject to corruption through a buffer overrun attack that can lead to code execution attacks. This recommended defense raises the bar substantially on the attackers.

Identify any long-lived pointers in your code. Access to these functions should be through encoded pointers using code like this:

```
// g_pFoo is a global point that points to foo

void g_pFoo = EncodePointer(&foo);

// Now get the encoded pointer

void *pFoo = DecodePointer(g_pFoo);
```

The global pointer (g_pFoo) is encoded during the initialization phase, and its true value remains encoded until the pointer is needed. Each time g_pFoo is to be accessed, the code must call DecodePointer.

- Fix code flagged by /W4 compiler warnings. Attackers are finding and exploiting more obscure classes of vulnerabilities as traditional stack and heap buffer overruns become harder to find. To this end, it is recommended that all W4 warning messages are fixed prior to release.
- No global exception handlers. Exceptions are a powerful way to handle run-time errors, but they can also be abused in a way that could mask errors or make it easier for attackers to compromise systems.
- Restrict database permissions. Only grant "execute" permission on all stored procedures, and grant that permission only for the application domain group. For example:
  - Run your online service as "Network Service," for example, *phx\$machinename*
  - Join that domain account to a domain group, for example, *phx\mywebappgroup*

  Ensure that the application database is set for execute permission only (and no other permissions) on its stored procedures and those permissions are associated with a domain group of which that application or server is a member. The principle of least privilege should be used and the group associated with the online service should not be granted access to any other object and no other user or group should be used within the online service for communication with the SQL server.
- NULL out freed memory pointers in new code. This helps reduce the severity of double-free bugs and bugs that overwrite "dangling" pointers. For example:

  ```
  char *p = new char[N];

  ...

  delete [] p;
  ```

  Add this statement after the delete operator:

  ```
  p = NULL;
  ```

  The same process applies to any dynamic allocation and freeing pattern (for example, using malloc/free, GlobalAlloc/GlobalFree, or VirtuaAlloc/VirtualFree).
- Lock ActiveX controls to a defined set of domains. Identify any ActiveX controls, new and existing, that can be locked to a preselected set of domains, and incorporate the SiteLock 1.15 Template for ActiveX Controls during implementation to lock each control to that set of domains. Note that each control can have its own set of domains.
- ClickJacking defense. For each page that could contain user controllable content, you should use a "frame-breaker" script and include the HTTP response header named X-FRAME-OPTIONS in each

authenticated page. The Watcher tool may be of use in meeting this recommendation. The exit criteria for this recommendation is as follows:

1. A "frame-breaker" script is included in each authenticated page to prevent unintentionally framing.
2. The X-FRAME-OPTIONS header has been added to all authenticated page HTTP responses that should not be framed (for example, DENY) or is utilized to only allow trusted sites to frame site content (for example, the current site with the use of SAMEORIGIN).

- COM best practices. Check for HRESULT misuse and uninitialized [PROP]VARIANTs. All HRESULTs should be set to valid COM values, (such as S_OK, S_FALSE, or E_FAILED) and all VARIANTs are initialized correctly.
- Restrict database permissions. Only grant Execute permission on all stored procedures, and grant that permission only for the application domain group. For example, run your online service as Network Service (for example, phx\$machinename) or join that domain account to a domain group (for example, phx\mywebappgroup).

  Make sure that this group is granted execute permissions only on your stored procedures. The principle of least privilege should be used and the group associated with the online service should not be granted access to any other object (and no other user or group should be used within the online service for communication with the SQL server).

- Use Transport Layer encryption securely. Properly use Transport Layer Security (TLS) when communicating with another entity, and verify that your service checks the **Common Name** attribute to be sure it matches the host with which you intended to communicate. Verify that your service consults a CRL for an updated list of revoked certificates at a frequent interval. If your service is accessible via a browser, verify that no security warnings appear at any visited URL for any supported browser.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 11: Stage 6—Secure Coding Policies, Chapter 19: SDL Banned Function Calls
- Compiler Security Checks in Depth
- SAL Annotations

Phase Four: Verification

During the Verification phase, you ensure that your code meets the security and privacy tenets you established in the previous phases. This is done through security and privacy testing, and a security push—which is a team-wide focus on threat model updates, code review, testing, and thorough documentation review and edit. A public release privacy review is also completed during the Verification phase.

Security and Privacy Testing

Security testing addresses two broad areas of concern:

- Confidentiality, integrity, and availability of the software and data processed by the software. This area includes all features and functionality designed to mitigate threats as described in the threat model.
- Freedom from issues that could result in security vulnerabilities. For example, a buffer overrun in code that parses data could be exploited in ways that have security implications.

Begin security testing very soon after the code is written. This testing stage requires one full test pass after the verification stage because potential issues and vulnerabilities might change during development.

Security testing is important to the Security Development Lifecycle. As Michael Howard and David LeBlanc note, in *Writing Secure Code, Second Edition*, "The designers and the specifications might outline a secure design, the developers might be diligent and write secure code, but it's the testing process that determines whether the product is secure in the real world."

Security Requirements

- Where input to file parsing code could have crossed a trust boundary, file fuzzing must be performed on that code. All issues must be fixed as described in the Security Development Lifecycle (SDL) Bug Bar. Each file parser is required to be fuzzed using a recommended tool.

  Win32/64/Mac: An Optimized set of templates must be used. Template optimization is based on the maximum amount of code coverage of the parser with the minimum number of templates. Optimized templates have been shown to double fuzzing effectiveness in studies. A minimum of 500,000 iterations, and have fuzzed at least 250,000 iterations since the last bug found/fixed that meets the SDL Bug Bar.

  WinCE and Xbox: 100,000 bug free iterations, since the last bug found/fixed that meets the SDL Bug Bar. All file fuzzing bugs must be filed and triaged according to the SDL Bug Bar's guidance.
- If the program exposes remote procedure call (RPC) interfaces, you must use an RPC fuzzing tool to test for problems. You can find RPC fuzzers on the Internet. This requirement applies only to

programs that expose RPC interfaces. All fuzz testing must be conducted using "retail" (not debug) builds and must correct all issues as described in the SDL Bug Bar.

- If the project uses ActiveX controls, use an ActiveX fuzzer to test for problems. ActiveX controls pose a significant security risk and require fuzz testing. You can find ActiveX fuzzers on the Internet. Conduct all fuzz testing using "retail" (not debug) builds, and correct all issues as described in the SDL Privacy Bug Bar (Sample) and SDL Security Bug Bar (Sample) appendices.
- Satisfy Win32 testing requirements as described in Appendix J: SDL Requirement: Application Verifier. The Application Verifier is easy to use and identifies issues that are MSRC patch class issues in unmanaged code. AppVerifier requires a modest resource investment and should be used throughout the testing cycle. AppVerifier is not optimized for managed code.
- Define and document the security bug bar for the product. Verify that a security bug bar has been established and approved. Vulnerabilities include:
  - Elevation of privilege (the ability either to execute arbitrary code or to obtain more privilege than intended).
  - Denial of service.
  - Targeted information disclosure (where the attacker can locate and read information from anywhere on the system, including system information, that was not intended or designed to be exposed).
  - Spoofing.
  - Tampering (permanent modification of any user data or data used to make trust decisions in a common or default scenario that persists after restarting the operating system or application).
- For online services and/or LOB applications, use approved cross-site scripting scanning test tools with the bug tracking system, and enter all vulnerabilities found into your bug tracking system. All vulnerabilities must be addressed prior to the Final Security Review.
- For online services and/or LOB applications that implement web services, use an approved scanner to check for XML parsing problems.
- Complete testing for kernel-mode drivers. The product team must complete the following testing for every kernel-mode driver:

  Driver Verifier

  1. Using Windows Vista or Windows Server 2008, complete a full functional test on the driver with Driver Verifier enabled using /standard mode.
  2. Execute all code paths in the driver with Driver Verifier enabled using /standard mode.

  Device Path Exerciser

  1. Run Device Path Exerciser specifically against each driver in the product (using the */dr* parameter).

2. Run Device Path Exerciser with Driver Verifier enabled.

   To meet the exit criteria, every kernel-mode driver in the product must pass the Driver Verifier and Device Path Exerciser tests. Driver Verifier is available in the Windows Driver Kit, see **Driver Development Tools -> Tools for Verifying Drivers -> Driver Verifier** or, on MSDN, see http://msdn.microsoft.com/en-gb/library/ff545448.aspx. Device Path Exerciser is available in the Windows Driver Kit, see **Driver Development Tools -> Tools for Testing Drivers -> Device Path Exerciser** or, on MSDN, see http://msdn.microsoft.com/en-gb/library/ff544851.aspx.

- COM object testing. Any product that ships a registered COM object must meet the following minimum criteria:
  1. COM objects must be compiled and tested with the SDL required switches enabled (for example, a COM object must be tested with NX and ASLR flags applied to the control and on a machine with NX and ASLR enabled).
  2. All methods in a COM object's supported interfaces must execute without access violations when called with valid data.
  3. COM objects must follow all published rules on reference counting. See the MSDN documentation on Addref and Release.
  4. COM objects must be tested for reliable query, instantiation, and interrogation by any COM container without returning an invalid pointer, leaking memory, or causing access violations.
  5. COM objects must follow the published rules for QueryInterface.
- (Web applications only) If a site provides any authenticated access, then the crossdomain.xml or clientaccesspolicy.xml files for the site must only allow specifically enumerated authorized sites (that is, no wildcards). When using JavaScript, do not set document.domain to a shared top-level domain (for example, microsoft.com). Use a more specific domain instead. Exit criteria is as follows:

  Read-Only Unauthenticated Sites and Services

  Sites and web services that do not require authentication and provide read-only information have no action items for this requirement. However, keep in mind that policy files are site-wide, so a policy meant for an unauthenticated site will also apply to any other sites on the same server. If the application is a public service that could be used in mashups, other web services, or Flash or Silverlight® applications and thus requires a permissive crossdomain.xml or accesspolicy.xml file (one allowing * or a broad top-level domain, like msn.com or live.com), then interactive websites or authenticated APIs may not be hosted on the same domain.

  Authenticated Websites

  If an application is a standard web UI (not a service) that hosts web services for its own use, or has Flash and Silverlight components on the site, any crossdomain.xml or clientaccesspolicy.xml file in the root directory must allow access *only* to the sites that contain the appropriate Flash and Silverlight components or web services.

Authenticated Web Services

If a site has functions available only to authenticated users but also needs to be accessed by a Flash or Silverlight application, ensure that any Flash or Silverlight applications that the site uses load the policy file only from the root directory of the site, and ensure that the value of does not set domain="*". In addition, if such a site must be accessed by Silverlight applications, ensure a clientaccesspolicy.xml that allows only the desired sites is present, since Silverlight does not honor Flash crossdomain.xml files with policies other than "*". Authenticated sites with Flash and Silverlight front-ends must always use crossdomain.xml or clientaccesspolicy.xml to restrict access, since an open policy (domain="*") will allow any Internet site the user visits to take action as the user.

JavaScript

Scripts setting document.domain to any value should be validated to ensure that:

1. The site checks that the caller is on a list of allowed sites before setting document.domain.
2. If the site deals with PII in any way, document.domain is not set to a top-level domain (for example, live.com) but only to an appropriate subdomain (for example, billing.live.com).

- Perform Application Verifier tests. Test all discrete applications within a shipping product for heap corruption and Win32 resource issues that might lead to security and reliability issues. You can detect these issues using AppVerifier, available at http://technet.microsoft.com/en-us/library/bb457063.aspx. Exit Criteria: All tests in the application's functional test suite have been run under AppVerifier, and all issues have been fixed.
- Network fuzzing. Fuzzing of network interfaces is one of the primary tools of security researchers and attackers, and network facing applications are arguably the most easily accessed target for a remote attacker. Each network parser must successfully handle 100,000 malformed packets without error.
- Binary analysis. If obfuscated binaries are being shipped, BinScope must be run on the pre-obfuscated version of each binary instead of the obfuscated version to ensure that issues were identified correctly.

Security Recommendations

- Create and complete security testing plans that address these issues:
  - Security features and functionality work as specified. Ensure that all security features and functionality that are designed to mitigate threats perform as expected.
  - Security features and functionality cannot be circumvented. If a mitigation can be bypassed, an attacker can try to exploit software weaknesses, rendering security features and functionality useless.
  - Ensure general software quality in areas that can result in security vulnerabilities. Validating all data input and parsing code against malformed or unexpected data is a common way attackers

try to exploit software. Data fuzzing is a general testing technique that can help prevent such attacks.

- Penetration testing. Use the threat models to determine priorities, test, and attack the software as a hacker might. Use existing tools or design new tools, if needed.
  - Hire third-party security firms as appropriate. Depending on the business goals for your project and availability of resources, consider engaging an external security firm for a security review or penetration testing.
- Develop and use vulnerability regression tests. If the code has ever had a security vulnerability reported, it is strongly suggested that you add regression tests to the test suite for that component to ensure that similar vulnerabilities are not inadvertently re-introduced to the code. Similarly, if there are other products with similar functionality in the market that have suffered publicly reported vulnerabilities, add tests to the test plan to prevent similar vulnerabilities.
- For online services and/or LOB applications, conduct data flow testing. Any externally accessible pages and interfaces must have tests. This should include pages that automatically redirect.
- Run through your test cases with WinHTTP, the debug version of wininet, or another application that captures all page transitions. Make sure that no part of the flow can be bypassed.
- If the feature exposes SOAP or DCOM interfaces or any other services, these must also be tested. Ensure that no step can be skipped or bypassed.
- If your feature requires authenticating a user before providing access, ensure that it is not possible to bypass this authentication step by directly connecting to the backend.
- For online services and/or LOB applications, conduct replay testing. Replay all messages for any scenario you are responsible for to ensure that the expected outcome occurs. For example, try and change the password and then repeat, or attempt to reuse security tokens in other contexts (for example, try using a login token in a password reset flow).
- For online services and/or LOB applications, cover input validation testing scenarios and variants. Do not do this through a web browser, since it will honor server-specified field lengths. Test cases must cover the following scenarios:
  - Random inputs. Ensure that a full range of ASCII and Unicode characters are used. All verification should be “allow” based instead of “block” based (deny everything that is not explicitly allowed).
  - Large inputs. Large strings should be attempted.
  - Script injection.
  - SQL injection.
  - Path traversal. Try and pass filenames, like ../../../../../../boot.ini, to bypass directory access controls.

- Malformed XML blobs. Attempt to submit XML that does not match the target schema, if your feature uses XSL attempt to pass XSL processing instructions within your input. Note that this is best done either using valid, but slightly incorrect, XML data to bypass the .NET validation code or disabling .NET validation checks before testing. All final release code to be used in production environments must not disable XML and other validation code in .NET.

- Secure Code Review. Security code reviews are a critical component of the Security Development Lifecycle. Given the opportunity to review old code or work on a new cool feature, developers lean towards the latter. Unsurprisingly, attackers don't target only new functionality; they will attack all code, regardless of its age. Waiting to make the code more secure in the next version of the product is not a good solution for protecting customers, and therefore, high-risk items (Critical) that are considered the most sensitive and important for security should be reviewed in depth at the earliest opportunity.

  Determine the most at-risk components (Critical) and perform an in-depth security review of the code making up those components. For critical components or if time allows, also review Important items. Use the following guidelines to determine the most at-risk components.

  1. Define the code review priority based on these criteria:

  - **Critical code** is considered to be the most sensitive from a security standpoint. The following are examples of Critical code, but please note this is not necessarily a definitive list. Critical code is all Internet- or network-facing code, code in the Trusted Computing Base (TCB)—such as kernel or SYSTEM code, code running as administrator or Local System, code running as an elevated user (also includes LocalService and NetworkService), or features with a prior history of vulnerability, regardless of version. Any code that handles secret data, such as encryption keys and passwords, is considered Critical code. For managed code, Critical code is considered to be any unverifiable code (any code that the standard PEVerify.exe tool reports as not verified). All code supporting functionality exposed on the maximum attack surface is considered Critical code by definition.
  - **Important code** is optionally installed code that runs with user privilege, or code that is installed by default that doesn't meet the Critical criteria.
  - **Moderate code** is rarely used code and setup code. *Setup code that handles secret data, such as encryption keys and passwords, is always considered Critical code.*
  - Any code or component with high rates of security bug discovery is considered to be Critical code, even if it otherwise maps to Important or Moderate per the previous definitions. While the definition of high rates is subjective within the team, it is important to examine the portions of code that have experienced the highest rates of security issues with extra scrutiny.
  - Don't forget to include and prioritize all sample code shipped with the product. While generalized guidelines are difficult, consider how customers will be using the samples. Samples that are

expected to be compiled and used with little changes in production environments should be considered Critical. "Hello World" applications are more likely to be considered Moderate code.

2. Identify development and testing owners for everything in products. The following are required to meet this security recommendation:

- All Critical source code should be thoroughly reviewed by inspection teams and code-scanning tools.
- All Important code should be reviewed using code-scanning tools and some human analysis.
- Development owners for all source code and testing owners for all binaries have been identified, documented, and archived.
- All source code is assessed and assigned a severity—Critical, Important, Moderate, or Low. This information is recorded in a document or spreadsheet and is archived.

- Use a passive security auditor. Use Watcher and Fiddler to detect vulnerabilities. Browse through every page in your web application (or run a prerecorded web macro that hits every page) with the Watcher plug-in for Fiddler enabled. If Watcher finds any potential vulnerabilities, you must fix them. Repeat this process until the run is completed with no flagged issues.

Privacy Recommendations

- For P1 and P2 projects, include privacy testing in your master test plan. Privacy testing of platform components deployed in organizations should include verification of organizational policy controls that affect privacy (these controls are listed in Appendix C: SDL Privacy Questionnaire). Privacy testing for features that transfer data over the Internet should include monitoring network traffic for unexpected network calls.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 12: Stage 7—Secure Testing Policies.
- *How to Break Software: A Practical Guide to Testing* (ISBN 978-0201796193; ISBN-10 0201796198).
- *How to Break Software Security: Effective Techniques for Security Testing* (ISBN 978-0321194336; ISBN-10 0321194330).

Security Push

A security push is a team-wide focus on threat model updates, code review, testing, and thorough documentation review and edit. A security push is not a substitute for a lack of security discipline. Rather, it is an organized effort to uncover changes that might have occurred during development, improve security in any legacy code, and identify and remediate any remaining vulnerabilities. However, it should be noted that it is not possible to build security into software with only a security push.

A security push occurs after a product has entered the verification stage (reached code/feature complete). It usually begins at about the time beta testing starts. Because the results of the security push might alter the default configuration and behavior of a product, you should perform a final beta test review after the security push is complete and after all issues and required changes are resolved.

It is important to note that the goal of a security push is to find vulnerabilities, not to fix them. The time to fix vulnerabilities is after you complete the security push.

Push Preparation

A successful push requires planning:

- You should allocate time and resources for the push in your project's schedule, before you begin development. Rushing the security push will cause problems or delays during the Final Security Review.
- Your team's security coordinator should determine what resources are required, organize a security push leadership team, and create the needed supporting materials and resources.
- The security representative should determine how to communicate security push information to the rest of the team. It is helpful to establish a central intranet location for all information related to the push, including news, schedules, plans, forms and documents, white papers, training schedules, and links. The intranet site should link to internal resources that help the group execute the security push. This site should serve as the primary source of information, answers, and news for employees during the push.
- There must be well-defined criteria to determine when the push is complete.

Your team will need training before the push. At a minimum, this training should help team members understand the intent and logistics of the push itself. Some members might also require updated security training and training in security or analysis techniques that are specific to the software that is undergoing the push. The training should have two components—the push logistics, delivered by a senior member of the team conducting the push, and technical and role-specific security training.

Push Duration

The amount of time, energy, and team-wide focus that a security push requires differs depending on the status of the code base and the amount of attention the team has given to security earlier in development. A security push requires less time if your team has:

- Rigorously kept all threat models up to date.
- Actively and completely subjected those threat models to penetrations testing.
- Accurately tracked and documented attack surfaces and any changes made to them.

- Completed security code reviews for all high-severity code (see discussion later in this section for details about how severity is assessed).
- Identified and documented development and testing contacts for all code released with the product.
- Rigorously brought all legacy code up to current security standards.
- Validated the security documentation plan.

The duration of a security push is determined by the amount of code that needs to be reviewed for security. Try to conduct security code reviews throughout development, after the code is fairly stable. If you try to condense too many code reviews into too brief a time period, the quality of code reviews suffers. In general, a security push is measured in weeks, not days. You should aim to complete the push in three weeks and extend the time as necessary.

Security Requirements

- Review and update threat models. Examine the threat models that were created during the Design phase. If circumstances prevented creation of threat models during Design phase, you must develop them in the earliest phase of the security push.
- Review all bugs that affect security against the security bug bar. Ensure that all security bugs contain the security bug bar rating.

Privacy Requirements

Review and update the SDL Privacy Questionnaire form (Appendix C to this document) for any material privacy changes that were made during the implementation and verification stages. Material changes include:

- Changing the style of consent.
- Substantively changing the language of a notice.
- Collecting different data types.
- Exhibiting new behavior.

Security Recommendations

- Conduct security code reviews for at-risk components. Use the following information to help determine which components are most at risk, and use this determination to set priorities for security code review. High-risk items (Critical) must be reviewed earliest and most in depth. For a minimal checklist for security issues to be aware of during code reviews, see "Appendix D: A Developer's Security Checklist" in *Writing Secure Code, Second Edition* (p. 731).
- Identify development and testing owners for everything in the program. Identify a development owner for each source code file. Identify a quality assurance owner for each binary file. Record this information in a document or spreadsheet and use a document/source tracking system to store it.

- Prioritize all code before you start the push. Track severity ratings in a document or spreadsheet that lists the development and quality assurance owners. Subject all code to the same criteria for prioritization, including legacy code. Many security vulnerabilities have come from legacy code that was created before the introduction of security pushes, threat modeling, and the other processes that are included in the Security Development Lifecycle.
- Ensure that you include and prioritize all sample code shipped with the product. Consider how users will use the samples. Samples that are expected to be compiled and used with small changes in production environments should be considered Critical.
- Re-evaluate the attack surface of the software. It is important to re-evaluate your team's definition of attack surface during the security push. You should be able to calculate the attack surface based on information described in the design specifications for the software. Measurement of the attack surface enables you to understand which components have direct exposure to attack and the highest risk of damage if a security breach occurs. Focus effort on areas of highest risk areas, and take appropriate corrective actions. These actions might include:
    - Prolonging the push for especially error-prone components.
    - Deciding not to ship a component until it is corrected.
    - Disabling a component by default.
    - Re-designating a component for future removal from the software (deprecating it).
    - Modifying development practices to make vulnerabilities less likely to be introduced by future modifications or new developments.

After you evaluate the attack surface, update attack surface documentation as appropriate.

- As time permits, consider code reviews for all components tagged with a severity level of Important.
- Review the security documentation plan. Examine how any changes to the product design during development have affected security documentation. Ensure that the security documentation plan meets all user needs.
- Focus the entire team on the push. When team members finish reviewing and testing their own components, they should help others in the group.

Code severity definitions are provided in the following list:

- Critical code is considered the most sensitive from a security standpoint. The following examples of Critical code are not necessarily a definitive list:
    - All Internet-facing or network-facing code.
    - Code in the Trusted Computing Base (TCB) (for example, kernel or SYSTEM code).
    - Code running as administrator or Local System.

- Code running as an elevated user (including LocalService and NetworkService).
- Features with a history of vulnerability, regardless of version.
- Any code that handles secret data, such as encryption keys and passwords.
- Any unverifiable managed code (any code that the standard PEVerify.exe tool reports as not verified).
- All code supporting functionality exposed on the maximum attack surface.

- Important code is optionally installed code that runs with user privilege or code that is installed by default that does not meet the Critical criteria.
- Moderate code is rarely used code and setup code. (Setup code that handles secret data, such as encryption keys and passwords, is always considered Critical code.)
- Any code or component that has experienced large numbers of security issues is considered Critical code, even if it would otherwise be considered Important or Moderate. Although the definition of large numbers is subjective, it is important to scrutinize carefully the portions of code that contain the most security vulnerabilities.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 13: Stage 8—The Security Push

Phase Five: Release

The Release phase is when you ready your software for public consumption and, perhaps more importantly, you ready yourself and your team for what happens once your software is in the hands of the user. One of the core concepts in the Release phase is planning—mapping out a plan of action, should any security or privacy vulnerabilities be discovered in your release—and this carries over to post-release, as well, in terms of response execution. To this end, a Final Security Review and privacy review is required prior to release.

Public Release Privacy Review

Before any public release (including alpha and beta test releases), update the appropriate SDL Privacy Questionnaire for any significant privacy changes that were made during implementation verification. Significant changes include changing the style of consent, substantively changing the language of a notice, collecting different data types, and exhibiting new behavior.

Although privacy requirements must be addressed before any public release of code, security requirements need not be addressed before public release. However, you must complete a Final Security Review before final release.

Privacy Requirements

- Review and update the Privacy Companion form.
  - For a P1 project, your privacy advisor reviews your final SDL Privacy Questionnaire (Appendix C to this document), helps determine whether a privacy disclosure statement is required, and gives final privacy approval for public release.
  - For a P2 project, you need validation by a privacy advisor if any of the following is true:
    - A design review is requested by a privacy advisor.
    - You want confirmation that the design is compliant with privacy standards.
    - You wish to request an exception.
  - For a P3 project, there are no additional privacy requirements.
- Complete the privacy disclosure.
  - Draft a privacy disclosure statement as advised by the privacy advisor. If your privacy advisor indicates that a privacy disclosure is waived or covered, you do not need to meet this requirement.
  - Work with your privacy advisor and legal representatives to create an approved privacy disclosure.
  - Post the privacy disclosure to the appropriate website before each public release.

Privacy Recommendations

- Create talking points as suggested by the privacy advisor to use after release to respond to any potential privacy issues.
- Review deployment guidance for enterprise programs to verify that privacy controls that affect functionality are documented. Conduct a legal review of the deployment guide.
- Create "quick text" for your support team that addresses likely user questions, and generally foster strong and frequent communication between your development and support teams.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 14: Stage 9—The Final Security Review

Planning

Any software can be released with unknown security issues or privacy issues, despite best efforts and intentions. Even programs with no known vulnerabilities at the time of release can be subject to new

threats that emerge and might require action. Similarly, privacy advocates might raise privacy concerns after release. You must prepare before release to respond to potential security and privacy incidents. With proper planning, you should be able to address many of the incidents that could occur in the course of normal business operations.

Your team must be prepared for a zero-day exploit of a vulnerability—one for which a security update does not exist. Your team must also be prepared to respond to a software security emergency. If you create an emergency response plan before release, you will save time, money, and frustration when an emergency response is required for either security or privacy reasons.

Security Requirements

- The project team must provide contact information for people who respond to security incidents. Typically, such responses are handled differently for products and services.
    - Provide information about which existing sustained engineering (SE) team has agreed to be responsible for security incident response for the project. If the product does not have an identified SE team, they must provide an emergency response plan (ERP) and provide it to the incident response team. This plan must include contact information for three to five engineering resources, three to five marketing resources, and one or two management resources who are the first points of contact when you need to mobilize your team for a response effort. Someone must be available 24 hours a day, seven days a week, and contacts must understand their roles and responsibilities and be able to execute on them when necessary.
    - Identify someone who is responsible for security servicing. All code developed outside the project team (third-party components) must be listed by filename, version, and source (where it came from).
    - You must have an effective security response process for servicing code that has been inherited or reused from other teams. If that code has a vulnerability, the releasing team may have to release a security update even though it did not develop the code.
    - You must also have an effective security response process for servicing code that has been licensed from third parties in either object or source form. For licensed code, you also need to consider contractual requirements regarding which party has rights and obligations to make modifications, associated service level agreements (SLAs), and redistribution rights for any security modifications.
- Create a documented sustaining model that addresses the need to release immediate patches in response to security vulnerabilities and does not depend entirely on infrequent service packs.
- Develop a consistent and comprehensible policy for security response for components that are released outside of the regular product release schedule (out-of-band) but that can be used to update or enhance the software after release. For example, Windows must plan a response to security

vulnerabilities in a component, such as DirectX, that ships as part of the operating system but that might also be updated independently of the operating system, either directly by the user or by the installation of other products or components.

- Disable tracing and debugging in ASP.NET applications prior to deployment. This neutralizes the following possible security vulnerabilities:
  - When tracing is enabled for the page, every browser requesting it also obtains the trace information that contains sensitive data about internal server state and workflow. This information could be security-sensitive.
  - When debugging is enabled for the page, errors happening on the server result in a full set of stack trace data presented to the browser. This data may expose security-sensitive information about the server's workflow.

Privacy Requirements

- For P1 and P2 projects, identify the person who is responsible for responding to all privacy incidents that may occur. Add this person's e-mail address to the Incident Response section of the SDL Privacy Questionnaire (Appendix C to this document). If this person changes positions or leaves the team, identify a new contact and update all SDL Privacy Questionnaire forms for which that person was listed as the privacy incident response lead.
- Identify additional development and quality assurance resources on the project team to work on privacy incident response issues. The privacy incident response lead is responsible for defining these resources in the Incident Response section of the SDL Privacy Questionnaire.
- After release, if a privacy incident occurs, you must be prepared to follow the SDL Privacy Escalation Response Framework (Appendix K to this document), which might include risk assessment, detailed diagnosis, short-term and long-term action planning, and implementation of action plans. Your response might include creating a patch, replying to media inquiries, and reaching out to influential external contacts.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 15: Stage 10—Security Response Planning
- Appendix K: SDL Privacy Escalation Response Framework (Sample)

Final Security Review and Privacy Review

As the end of your software development project approaches, you need to be sure that the software is secure enough to ship. The Final Security Review (FSR) helps determine this. The security team assigned to the project should perform the FSR with help from the product team to ensure that the software complies

with all SDL requirements and any additional security requirements identified by the security team (such as penetration testing or additional fuzz testing).

A Final Security Review can last anywhere from a few days to six weeks, depending on the number of issues and the team's ability to make necessary changes.

It is important to schedule the FSR carefully—that is, you need to allow enough time to address any serious issues that might be found during the review. You also need to allow enough time for a thorough analysis; insufficient time could cause you to make significant changes after the FSR is completed.

The FSR Process

- Define a due date for all project information that is required to start the FSR. To minimize the likelihood of unexpected delays, plan to conduct an FSR four to six weeks before release to manufacturing (RTM) or release to web (RTW). Your team might need to revalidate specific decisions or change code to fix security issues. The team must understand that additional security work needs to be performed during the FSR.
- The FSR cannot begin until you have completed the reviews of the security milestones that were required during development. Milestones include in-depth bug reviews, threat model reviews, and running all SDL-mandated tools.
    - Reconvene the development and security leadership teams to review and respond to the questions posed during the FSR process.
    - Review threat models. The security advisor should review the threat models to ensure that all known threats and vulnerabilities are identified and mitigated. Have complete and up-to-date threat models at the time of the review.
    - Review security issues that were deferred or rejected for the current release. The review should ensure that a consistent, minimum security standard was adhered to throughout the development cycle. Teams should already have reviewed all security issues against the criteria that were established for release. If the release does not have a defined security bug bar, your team can use the standard SDL Security Bug Bar.
    - Validate results of all security tools. You should have run these tools before the FSR, but a security advisor might recommend that you also run other tools. If tool results are inaccurate or unacceptable, you might need to rerun some tools.
    - Ensure that you have done all you can to remove vulnerabilities that meet your organization's severity criteria so that there are no known vulnerabilities. Ultimately, the goal of SDL is to remove security vulnerabilities from products and services. No software release can pass an FSR with known vulnerabilities that would be considered as Critical, Important, Moderate, or Low.

- Submit exception requests to a security advisor for review. If your team cannot meet a specific SDL requirement, you must request an exception. Typically, such a request is made well in advance of the FSR. A security advisor reviews these requests and, if the overall security risk is tolerable, might choose to grant the exception. If the security risk is not acceptable, the security advisor will deny the exception request. It is best to address all exception requests as soon as possible in the development phase of the project.

Possible FSR Outcomes

Possible outcomes of an FSR include:

- **Passed FSR.** If all issues identified during the FSR are corrected before RTM/RTW, the security advisor should certify that the project has successfully met all SDL requirements.
- **Passed FSR (with exceptions).** If all issues identified during the FSR are corrected before RTM/RTW *or* the security advisor and team can reach an acceptable compromise about any SDL requirements that the project team was unable to resolve, the security advisor should identify the exceptions and certify that all other aspects of the project have successfully met all SDL requirements.
  - All exceptions and security issues not addressed in the current release should be logged and then addressed and corrected in the next release.
- **FSR escalation.** If a team does not meet all SDL requirements, and the security advisor and the product team cannot reach an acceptable compromise, the security advisor cannot approve the project, and the project cannot be released. Teams must correct whatever SDL requirements that they can or escalate to higher management for a decision.
  - Escalations occur when the security advisor determines that a team cannot meet the defined requirements or is in violation of an SDL requirement. Typically, the team has a business justification that prevents them from being compliant with the requirement. In such instances, the security advisor and the team should work together to compose a consolidated escalation report that outlines the issue—including a description of the security or privacy risk and the rationale behind the escalation. This information is typically provided to the business unit executive and the executive with corporate responsibility for security and privacy, to aid decision-making.
  - If a team fails to follow proper FSR procedures—either by an error of omission or by willful neglect—the result is an immediate FSR failure. Examples include:
    - Errors of omission, such as failure to properly document all required information.
    - Specious claims and willful neglect, including:
      - Claims of "Not subject to SDL" contrary to evidence.
      - Claims of "FSR pass" contrary to evidence, and software RTM/RTW without the appropriate signoff.

Such an incident can result in very serious consequences and, as such, should always and immediately be escalated to the project team executive staff and the executive in charge of security and privacy.

Security Requirements

- The project team must provide all required information before the scheduled FSR start date. Failure to do so may delay completion of the FSR. If the schedule slips significantly before the FSR begins, contact the assigned security advisor to reschedule.
- After the FSR is finished, the security advisor either signs off on the project as is or provides a list of required changes.
- For online services and/or LOB applications, projects releasing services are required to have a security score of B or above to successfully pass the FSR. Both Operations and Product groups are responsible for compliance. A product's security is managed at many levels. Vulnerabilities, whether in code or at host level, put the entire product (and possibly the environment) at risk.

Privacy Requirements

- Repeat the privacy review for any open issues that were identified in the pre-release privacy review or for material changes made to the product after the pre-release privacy review. Material changes include modifying the style of consent, substantively revising the language of a notice, collecting different data types, or exhibiting new behavior. If no material changes were made, no additional reviews or approvals are required.
- After the privacy review is finished, your privacy advisor either signs off on the product as is or provides a list of required changes.

Security Recommendations

- Ensure the product team is constantly evaluating the severity of security vulnerabilities against the standard that is used during the security push and FSR. Otherwise, a large number of security bugs might be reactivated during the FSR.

Resources

- *The Security Development Lifecycle* (ISBN 9780735622142; ISBN-10 0-7356-2214-0), Chapter 16: Stage 11—Product Release

Release to Manufacturing/Release to Web

Software release to manufacturing (RTM) or release to web (RTW) is conditional to completion of the Security Development Lifecycle process as defined in this document. The security advisor assigned to the release must certify that your team has satisfied security requirements. Similarly, for all products that have

at least one component with a privacy impact rating of P1, your privacy advisor must certify that your team has satisfied the privacy requirements before the software can be shipped.

Security Requirements

- To facilitate the debugging of security vulnerability reports and to help tools teams research cases in which automated tools failed to identify security vulnerabilities, all product teams must submit symbols for all publicly released products as part of the release process. This requirement is needed only for RTM/RTW binaries and any post-release binaries that are publicly released to users (such as service packs or updates, among others).
- Design and implement a sign-off process to ensure security and other policy compliance before you ship. This process should include explicit acknowledgement that the product successfully passed the FSR and was approved for release.

Privacy Requirements

- Design and implement a sign-off process to ensure privacy and other policy compliance before you ship. This process should include explicit acknowledgement that the product successfully passed the FSR and was approved for release.

Resources

N/A

Post-SDL Requirement: Response

Security Servicing and Response Execution

After a software program is released, the product development team must be available to respond to any possible security vulnerabilities or privacy issues that warrant a response. In addition, develop a response plan that includes preparations for potential post-release issues.

Resources

- Appendix K: SDL Privacy Escalation Response Framework (Sample)

Security Development Lifecycle for Agile Development

This section 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 section does not explain all the nuances of the SDL. To gain a deeper understanding of the SDL, you can review the main section of this document.

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

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.

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 Tooling and Automation later in this Agile-SDL section).

- 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 P.

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 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 Q: SDL-Agile Bucket Requirements.

Verification Tasks	Design Review	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 Q: SDL-Agile Bucket Requirements.

In this example, a team would be required to complete one verification requirement, one design review requirement, and one 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 section. 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 R of this document. Figure 2 provides an illustration of this process in action.

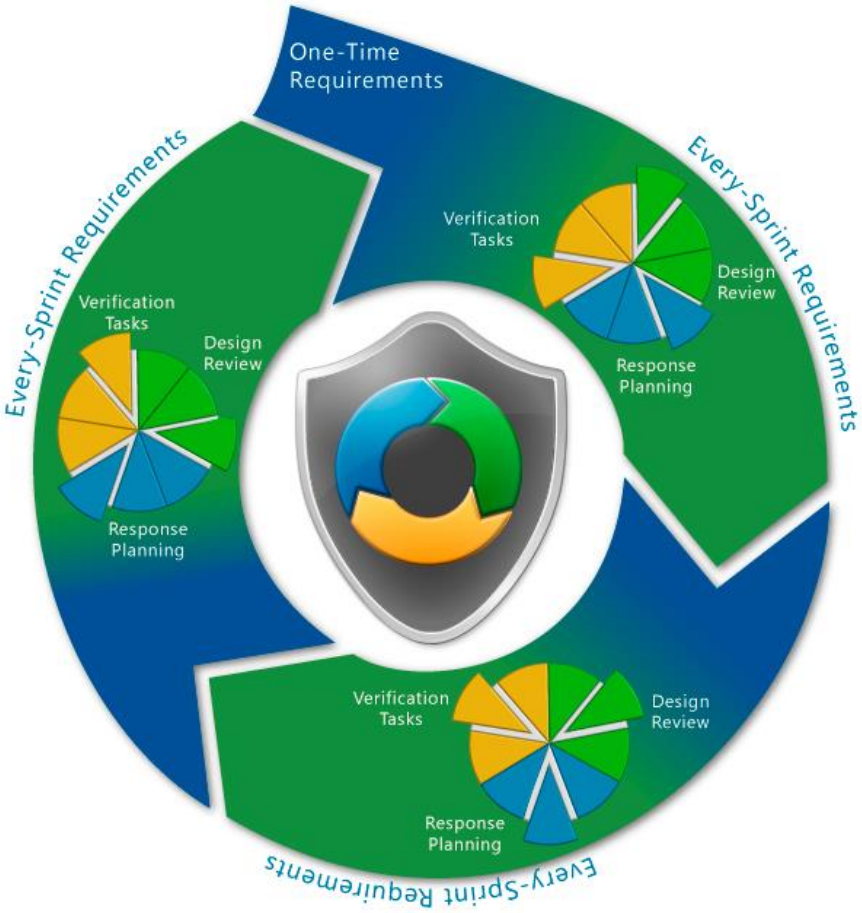

Figure 2. SDL-Agile process

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 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:

- **Fix issues identified by static code analysis tools for unmanaged code.** Use the currently required (or later) version of /analyze compiler option in Microsoft Visual Studio with the C and C++ compiler for native code for the target platforms and fix all required minimum (Min SDL) warnings. (Note: Also called /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.

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

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 website.

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 S: SDL-Agile 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 S: SDL-Agile 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 Moderate, 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 Low, 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 Important 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 Important 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, and identifying a privacy incident response person). 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

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

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.

Security Development Lifecycle for Line-of-Business Applications

The Security Development Lifecycle for Line-of-Business applications (SDL-LOB) defines the standards and best practices for providing security and privacy for new and existing line-of-business (LOB) applications currently under development or being planned for development. The SDL-LOB provides a mainstream approach to the SDL that serves line-of-business applications with additional requirements and recommendations. LOB applications are a set of critical computer applications that are vital to running an enterprise, such as accounting, human resources (HR), payroll, supply chain management, and resource planning applications. This guidance is positioned exclusively for **LOB applications or web applications and not for ISV/rich-client and server application development.**

Note: The goal of this section is to supplement the main SDL document and allow you to tailor a process specific to your LOB applications while meeting SDL requirements. If you don't see specific guidance for a particular task in the SDL-LOB, the guidance in the main SDL section is assumed to be in effect. To refer back to a specific phase within the main SDL, click the icon next to each phase heading throughout the SDL-LOB section.

To ensure minimal impact, the SDL-LOB overlays high-level security tasks against the standard SDL phases, as listed in the chevrons in Figure 3.

Figure 3. Standard SDL phases

The following table highlights LOB-specific tasks for each phase of the SDL. These tasks are in addition to those outlined in the main SDL portion of this document. Each task in the table is discussed by phase in the remainder of the LOB section. Note that the Response phase is not included in the table because there are no additional tasks required for that phase beyond what is discussed in the main SDL.

Training	Requirements	Design	Implementation	Verification	Release
LOB-specific training	Risk assessment • Application portfolio • Application risk assessment • Determine service level	Asset-centric threat modeling • Threat model • Design review	Internal review • Incorporate security checklists and standards • Conduct "self" code review • Security code analysis	Pre-production assessment • Comprehensive security Assessment • Bug tracking	Post-production assessment • Host level scan

It is important to note that organizations should adapt rather than adopt the Microsoft SDL-LOB process. Organizations are unique and should expect and plan for differences in resources, executive support, and security expertise.

The SDL-LOB, in various incarnations, has been in use since 2001 to identify and reduce risk for over 2,400 separate Microsoft LOB applications/releases.

Resources

- Visit the Information Security page for information on the Microsoft Information Security group, which is responsible for security risk management for Microsoft LOB applications.
- Appendix V: Lessons Learned and General Policies for Developing LOB Applications

Pre-SDL Requirements: Security Training for LOB

In this section and in the remainder of the SDL-LOB, only supplements to the original SDL are highlighted. To create a complete security plan for LOB applications, you should consult each section of the main SDL and the supplemental information contained in each phase of the SDL-LOB.

In addition to the basic concepts outlined in the main SDL, LOB training should include the following additional topics:

Basic Concepts

- **Secure design**, including the following topics:
  - Authentication.
  - Authorization.

- Asset handling.
- Auditing and logging.
- Secure communication. The HTTP data for web applications travels across networks in plain text and is subject to network eavesdropping attacks. This also applies to client-to-server and server-to-server communication.

- **Secure coding**, including the following topics:
  - Integer overflow/underflow.
  - Input validation and handling.
- **Regulatory**, which can include the following topics:
  - Compliance with SOX, HIPAA, GLBA, PCI.

Resources

- Security Training: See Securing Applications on MSDN
- SDL Quick Security Reference: SQL Injection

Phase One: Requirements for LOB

The Risk Assessment section that follows is exclusive to designing a security plan for LOB applications. It includes information on completing an application portfolio, assessing application risk, and determining service levels.

Risk Assessment

The Risk Assessment phase captures general security and privacy ''qualities'' to determine the appropriate amount of oversight. During this phase, an application is assessed to understand the potential risk it creates. If an application is high risk, it receives more oversight in the SDL-LOB process. If an application is low risk, it receives less oversight in the SDL-LOB process. The application team and application security team work in partnership to complete this phase.

When an application team proposes a new application or updates to an existing application, a risk assessment is completed. Application teams understand they must complete this step as a prerequisite for installing the application in a supported production environment. This risk assessment produces repeatable guidance on the type of oversight the project will receive in the SDL-LOB process.

There will be a close collaboration between the security and privacy subject-matter experts (SMEs) and the risk management/governance team for your organization. Risk management helps identify business objectives and therefore guidance for evaluating the risk posed by individual line-of-business applications. Risk management also affects and influences guidelines for rating the risk posed by individual vulnerabilities and classes of vulnerabilities filed during internal review and verification phases. For more information, see the Microsoft Security Risk Management Guide.

Security Requirements

- Application portfolio
  - Application teams enter application details in an application portfolio system that is used to track the life cycle of LOB applications within the enterprise.
  - The portfolio system can track information, such as contacts, dependencies, version history, deployment considerations, milestones, testing information and history, locations of relevant documents, and tasks and security controls used during the applications life cycle. Ideally, the portfolio would feature support, such as automated notification if the application is not in compliance with required (and, as appropriate—optional) controls. If you have a dedicated security team, the portfolio would also track the security SME assigned to perform an assessment, assessment history, artifacts, and, if appropriate, actual bugs.
  - Development teams must enter a new entry for the application if a new version of the application is being released so that it can follow this process cycle again.
- Application risk assessment
  - Application risk level is determined based on a questionnaire filled out by the application team. This determines the SDL-LOB tasks the application owner must complete and is used to determine if the application is in scope for a security and privacy assessment. Please see Appendix U: SDL-LOB Risk Assessment Questionnaire for more details.

    Note: The risk posed by an application may increase or decrease between successive releases and should be evaluated accordingly.

Mapping Risk to Security SME Service Levels

The output of the risk assessment dictates the degree of oversight from a security SME. Questions in the assessment are weighted together into an overall "score," while questions may dictate a review regardless of the overall "score." The sample questionnaire provided gives some guidance in this respect, but you will need to tailor it for your specific business and customer needs. This approach ensures consistency and reputability combined with flexibility. Experience has shown that a "one-size-fits-all" approach is not effective.

For example, based on application risk (High/Medium/Low), applications are serviced accordingly during various phases of the development. Note that all applications require oversight; however, application teams are still responsible for compliance with your implementation of the SDL-LOB (including appropriate requirements and recommendations from main SDL described earlier in this document).

Mapping of risk level to service levels are depicted in the following table:

Application Risk Level	Application Service Level
High	• Threat model • Design review Comprehensive review, which includes most or all of the following: • Code review (white box) • Penetration test (black box) • Privacy review • Deployment review
Medium	• Threat model (recommended) • Code review (white box) • Privacy review (as appropriate) • Deployment review (as appropriate)
Low	• Threat model (as appropriate) • Deployment review (as appropriate)

- Application risk level. The application risk level is based on the type of data managed by the system, the system functionality, and the environmental controls. Sample framework for determining risk level is as follows:
  - **High-risk applications.** Internet-facing applications that handle personal data, other highly sensitive data, or executes sensitive/critical functions.
  - **Medium-risk applications.** Internal-facing applications that handle sensitive data (not highly sensitive) or execute important functions that are not critical.
  - **Low-risk applications.** Applications that manage publicly available data or execute functions that do not play a key role.
- Application service level
  - **Threat model.** The application team needs to create (or update) a threat model for the application. Threat modeling is, in some sense, a discovery process wherein you look at your application through a lens of security and/or privacy. The application team should own the creation of the threat model in consultation with a security/privacy SME. **Note:** Threat models are recommended, but compliance is not typically enforced for low and sometimes medium risk applications.
  - **Design review.** Similar to and distinct from threat modeling, a design review addresses additional issues, such as reviewing your design approach to critical areas in your application, including

authentication, authorization, input/data validation, exception management, user provisioning/deprovisioning, and other areas. Finally, you conduct a tier-by-tier component analysis and examine the security mechanisms employed by your key components, such as your presentation layer, business layer, and data access layer.

- **Comprehensive assessment.** This effort is the exhaustive testing of the application and the vulnerabilities found in the code base in an attempt to determine the exploitability of the issues found. Severity of findings can be influenced by the results of exploit testing. This service may include all or some of the following tasks as appropriate:
  - Code review (white box)
  - Penetration test (black box)
  - Regulatory compliance (privacy review)
  - Deployment review
- **Code review (white box).** Conducted to determine how many code vulnerabilities exist. Severity of findings is based on deviation from policy, standards, and best practices. The review should balance a line-by-line code inspection against prioritizing sensitive parts of the application, such as authentication, authorization, handling of sensitive data, and avoiding common security vulnerabilities, such as poor input validation, SQL injection, and failure to properly encode web output.

  In addition, the code review should verify compliance with security/privacy standards and policies. Violations of standards and policies are viewed as must-fix, Critical vulnerabilities.
- **Penetration test (black box).** Uses a mix of tools, such as wire sniffers and scanners, combined with actually running the application to verify expected and unexpected behavior.
- **Deployment review.** Executed against production environments to ensure primarily that access control and architectural issues conform to requirements; this niche serves double duty. This service level is a good starting point for applications that do not immediately warrant the two higher levels. These are often internal medium risk applications.
- **Privacy review.** Ensure the application complies with corporate, domestic, and international privacy requirements to prevent malicious monitoring of behavior, obtaining sensitive information, or identity theft.

Security Recommendations

- Visual Studio .NET Team System (or equivalent) can be used for bug tracking and management purposes.

- Dedicated security and privacy subject matter experts assist the application team during application development. These SMEs serve as resources for conducting all of the SDL-LOB tasks but, in particular, help perform specific tasks, such as a code reviews and penetration tests, among others.

Resources

- Privacy home page.
- Microsoft Operation Framework Deliver Phase provides guidance for getting operational concerns reflected during the Requirements phase of project development as well as getting release readiness in place as a validation step prior to production.
- Governance, Risk, and Compliance Service Management.

Phase Two: Design for LOB

The Design phase is crucial to ensure that the application is "secure by design" and compliant with security and privacy policies and standards. As with the standard SDL, threat modeling is crucial to accomplishing this, although the SDL-LOB distinguishes itself by taking a more asset-centric approach to creating the threat model. Threat modeling evaluates the threats and vulnerabilities that exist in the project's environment or that result from interaction with other systems. You cannot consider the Design phase complete unless you have a threat model or models that include such considerations. Threat models are critical components of the Design phase and reference a project's functional and design specifications to describe vulnerabilities and mitigations.

Threat Modeling and Design Review

During the Design phase of development, carefully review security requirements and expectations to identify security concerns. It is efficient to identify and address these concerns and risks during the Design phase rather than later in the development life cycle.

Threat modeling and conducting design reviews is a systematic process that is used to identify threats and vulnerabilities in the application. You must complete threat modeling during project design. A team cannot build a secure application unless it understands the assets the project is trying to protect, the threats and vulnerabilities introduced by the project, and details of how the project will mitigate those threats.

Important Design phase tasks include the following:

- **Threat models** are critical components of the Design phase and reference a project's functional and design specifications to describe vulnerabilities and mitigations.
- **A design review** of the high-risk LOB applications with a security SME.
- **A review** of both the threat model and design review with a security SME to ensure both the completeness and quality of outputs from both exercises.

Threat modeling is typically done by the application team but can be done in conjunction with a security SME. Design reviews are typically conducted with a security SME.

Threat Modeling

Threat modeling is one of the most effective ways to build security into the application development process. It makes the application less vulnerable to potential threats by identifying them before the application is built. This proactive process is the most important phase of the SDL-LOB because it reduces the reliance on reactive processes that depend either on penetration testing or user discovery of security vulnerabilities.

Choosing the Right Threat Modeling Tool for LOB Applications

Threat modeling can be done in a variety of ways using either tools or documentation/specifications to define the approach. An asset-centric approach to threat modeling is recommended for LOB applications.

- The Threat Analysis and Modeling Tool (TAM) is an asset-focused tool designed for LOB applications. It is used for applications for which business objectives, deployment pattern, and data assets and access control are clearly defined. The focus of the tool is to understand the business risk in the application, help identify controls needed to manage that risk, and protect the assets.
- The SDL Threat Modeling Tool is a software-focused tool designed for rich client/server application development (for example, Windows and SQL Server, among others). The tool assumes the final deployment pattern of the product is unknown (that is, if it will be used to manage business-critical applications with customer credit cards or not), so the focus of the tool is to ensure security of the software's underlying code.

The following needs are to be met when choosing a threat modeling approach for LOB applications:

- **Provide a consistent methodology for objectively identifying and evaluating threats to software applications.** In order for a threat modeling methodology to be practical, it needs to be consistently reproducible. Given the same input to the methodology, the output should remain unchanged.
- **Translate technical risk to business impact.** There are many forms of technical risks that can materialize in LOB applications of varying scope. Some examples might be a weak authentication protocol identified during architecture or the lack of input validation on an entry point identified during development. Such technical attributes, for example, of a software application can materialize

specific technical risks, such as HTTP replay attacks or SQL injection attacks. But the greater problem is not the technical risk; the primary problem is the (negative) business impact that may potentially be realized through these technical risks.

- **Empower the business to manage risk.** Security is all about risk management, and risk management essentially entails identification of risks and how those risks are managed. The most common forms of risk management are acceptance, avoidance, transference, and reduction. However, before business groups can make decisions on their risk management approach, they need to be empowered with the right information to make the most justifiable decision in terms of business needs.
- **Create awareness between teams of security dependencies and security assumptions.** The creation of a LOB application goes through many phases of a development life cycle. Some examples are requirements, design, development, verification, and deployment. Just as the business requirements need to be maintained during the entire development life cycle, so do the identified countermeasures. By having a standard documentation of a security strategy, it enables the application groups to create and maintain awareness between various teams (for example, the design team or the test team) of the security dependencies and security assumptions made during various phases of the development life cycle that will lead to the realization of the identified countermeasures.

Use of the SDL Threat Modeling Tool is discussed earlier in this document and the remainder of this section focuses exclusively on the TAM tool.

Security Requirements

Threat models should be completed for all applications, regardless of risk level.

- Ensure that all threat models meet minimal threat model quality requirements. That is, all threat models must contain digital assets or data, business objectives, components, and role information. It must have application use cases, data flow, call flows, generated threats, and mitigations. Threat model reports generated are consumed by the development team as actionable items. A threat model that is not actionable (in terms of selecting countermeasures and prioritizing by risk) is an incomplete threat model.
- All threat models and referenced mitigations should be reviewed and approved by the security SME. Ask architects, developers, testers, program managers, and others who understand the software to contribute to threat models and to review them. Solicit broad input and reviews to ensure the threat models are as comprehensive as possible.
- Threat model data and associated documentation (functional/design specifications) have been stored within the application portfolio system described previously or by using the document control system used by product team for archiving purposes.

Design Reviews

Conduct design reviews of high-risk applications by a security SME to ensure that the design conforms to security/privacy standards and policies. The advantages of this include:

- An architecture and design review helps you validate the security-related design features of your application before you start the development phase. This allows you to identify and fix potential vulnerabilities before they can be exploited and before the fix requires a substantial reengineering effort. Essentially this results in a reduced attack surface exposed by applications, thus increasing the security of the user and the system.
- Important design areas to be reviewed during this task are:
  - Deployment and infrastructure considerations
  - Input validation
  - Authentication
  - Authorization
  - Configuration management
  - Sensitive data
  - Session management
  - Cryptography
  - Parameter manipulation
  - Exception management
  - Auditing and logging
  - User provisioning/de-provisioning
  - Tier-by-tier analysis; walk through the logical tiers of your application, and evaluate security choices within your presentation, business, and data access layers
  - Application life cycle, including end-of-life requirements
  - Compliance with security/privacy standards and policies, in addition to regulatory requirements

There is a certain degree of overlap for some of these requirements and a threat model. Therefore the SME will defer, as necessary, to the artifacts created in the threat modeling process.

Security Recommendations

In addition to the specific security recommendation in the SDL for threat modeling, perform the following:

- Security issues identified during the design review task should be logged under projects bug tracking system.

Resources

- Threat modeling process: This is the home page for the asset-centric Threat Analysis and Modeling tool and related content.
- *Improving Web Application Security: Threats and Countermeasures*—Chapter 5: Architecture and Design Review for Security. While written for both .NET and web applications, this provides guidance that can be used for a variety of applications and technologies.

Phase Three: Implementation for LOB

For the LOB-SDL, additional tasks beyond the standard SDL Implementation phase include an internal review, which incorporates security checklists and standards, a self-directed code review, and code analysis.

Internal Review

The internal review is conducted by the application team.

Incorporate Security Checklist and Review Policies

- Tools. A mix of freeware, third-party tools to perform code analysis, or penetration testing can be employed during this phase. The challenge tools present is that they often require security expertise to filter false positives or to maximize the results from the tool. This is especially true for more sophisticated toolsets.
- Security checklist. Development teams must review available information resources to adopt appropriate coding techniques and methodologies. A coding checklist that describes the minimal requirements for any checked-in code for ASP.NET version 2.0 applications. See checklist items from the Security Checklist Index from Microsoft Patterns and Practices.
- Review internal policies and standards. Review internal policies created by the central security and policy team around application development and hosting to ensure that development teams are in sync with the security policies relevant to their application design. These policies may also reflect domestic and international legal requirements.
- Review and develop deployment guidelines. The application team needs to answer this question: How will the application be securely deployed removing artifacts, test code, and settings that are needed during development and testing? For example, web.config files in ASP.NET often contain the following statement to facilitate debugging during this phase.

```
<compilation debug="true">

<trace enabled="true"/>
```

However, it needs to be explicitly turned off in production, otherwise a malicious user may be able to take advantage of these settings to profile the application that could lead to an actual exploit. Whether there is a manual or automated process, the development team needs to work with the owners of the production servers to ensure that inappropriate code, artifacts, and settings are not actually used in production.

- Encrypt all secrets in text format whether in source code, web.config, machine.config, or *any* file. Unencrypted, these secrets are prone to discovery and can easily allow a partially compromised application or system to be further exploited, thus increasing the impact of the exploit. All passwords, connection strings with passwords, encryption keys, credentials, and other secrets are stored encrypted when stored in configuration files. Secrets must never be in source code encrypted or plaintext.

  Use one or more of the following as a countermeasure:

  - RSAProtectedConfigurationProvider This is the default provider and uses RSA public key encryption to encrypt and decrypt data. It can also be used across multiple machines (for example web.config file secrets in a cluster). See How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA.
  - DPAPIProtectedConfigurationProvider. This provider uses the Windows Data Protection API (DPAPI) to encrypt and decrypt data. See How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI.
  - In a single server deployment scenario, utilities such as Aspnet_setreg.exe are available to protect secrets. Also see the Knowledge Base (KB) article How to use the ASP.NET utility to encrypt credentials and session state connection strings (http://support.microsoft.com/kb/329290). .NET Framework 2.0 introduced a protected configuration feature that can be used to encrypt sensitive configuration file data using a command line tool Aspnet_Regiis.exe. How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA explains how to use the Aspnet_Regiis.exe tool with the RSAProtectedConfigurationProvider to encrypt sections of the configuration file.
    - **(New for SDL 5.2)** For multiple servers and clusters, use Distributed Key Manager (DKM) (preferred) or RSA.
    - **(New for SDL 5.2)** Privacy.NET Framework 2.0 introduced a protected configuration feature that can be used to encrypt sensitive configuration file data using the command line tool Aspnet_Regiis.exe. How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA explains the process of using the Aspnet_Regiis.exe tool with the **RSAProtectedConfigurationProvider** to encrypt sections of the configuration file.
- Input validation. Input validation must be applied at all identified entry points (including form fields, QueryStrings, cookies, HTTP headers, and web service parameters). It is a security practice never to trust user input data without first confirming its authenticity. Verify that all inputs to the application are validated on the server side before consuming. String data is validated for length and format.

Where possible use regular expressions to validate format. Make sure numeric data is validated for range (upper and lower bound) and type (signed vs. unsigned). String data should preferably use inclusion list (known, valid, and safe input) rather than exclusion list (rejecting known malicious or dangerous input).

- Line of Business Secure Code Review. High-risk items (Pri 1) that are considered the most sensitive or important for security should be reviewed in depth at the earliest opportunity. Resolve any Severity 1 bugs or bugs that are, per the Code Review Checklist, a standards violation. If the bug cannot be fixed, document the underlying infrastructure or framework limitation. Resolve all bugs that are moderate or higher based on the SDL Bug Bar. Development owners for all source code and testing owners for all binaries have been identified, documented, and archived—for example, in the source tracking system used by the product. All source code is assessed and assigned a priority—Pri 1, Pri 2, and Pri 3. This information is recorded in a document or spreadsheet and is archived in a tracking bug for ease-of-discovery.
- Conduct Internal Security Design Review. An architecture and design review helps you validate the security-related design features of your application before you start the development phase. Enter all security design requirements in the bug tracking system, triage the defects with the team and security advisor, correct all defects that are Severity 1 or higher, update threat models and work items accordingly.
- Secure sensitive data-at-rest. High business impact (HBI) data needs to be encrypted "at-rest," that is, when located in your database server or other data stores. Ensuring that HBI data is encrypted with appropriate encryption algorithms and key strength when at-rest in a data-store, and that this encryption takes place within the data store. Where possible, leveraging existing operating system or server functionality to manage encryption rather than building such functionality from scratch. Be sure that that this encryption actually secures sensitive data. For SQL Server, use SQL Server Transparent Data Encryption to encrypt the entire database. Note that this approach still requires good management of any keys used for encryption. Use T-SQL functions and SQL Server infrastructure to encrypt specific data elements.
- Secure sensitive data-in-transit. Encryption should be used for any HBI data that is transmitted over wired or wireless connections on the corporate network (corpnet) or is extranet/Internet-facing. Encryption should be used for any medium business impact (MBI) information in transit via wireless or wired connections when in-transit across the Internet.

Conduct 'Self' Code Review

The development team carries out 'self' code reviews of the application source code to detect low-hanging security vulnerabilities. See How to Perform a Security Code Review for Managed Code (Baseline Activity).

Run Code Analysis Tools and Incorporate Security Libraries

Use static analysis and runtime security tools. As appropriate, incorporate security libraries. The following is a list of free tools available from Microsoft for download.

Security Requirements

- Microsoft Anti-Cross-Site Scripting Library V4.2. Incorporate Anti-XSS library to protect ASP.NET web-based applications from XSS attacks. This library offers a more rigorous "white-list" approach than the native encoding methods found in .NET. Also featured is new support for globalization also not present in the .NET library.

- CAT.NET. Run CAT.NET on managed code (C#, Visual Basic .NET, J#) applications. CAT.NET is a snap-in to the Visual Studio IDE that helps you identify exploitable code paths for security vulnerabilities, such as Cross-Site Scripting - SQL Injection - Process Command Injection - File Canonicalization - Exception Information - LDAP Injection - XPATH Injection - Redirection to User Controlled Site.

- FxCop. FxCop is an application that analyzes managed code assemblies (code that targets the .NET Framework common language runtime) and reports information about the assemblies, such as possible design, localization, performance, and security improvements.

- Microsoft Source Code Analyzer for SQL Injection. Run this static code analysis tool that helps identify SQL injection vulnerabilities in Active Server Pages (ASP) code.

Security Recommendations

Security vulnerabilities identified during self-review and through code analysis tools should be logged under the project's bug tracking system.

Resources

- Index of Security Checklists. While focused on .NET and web application development, much of the guidance here is technology agnostic.
- Perform a Security Code Review for Managed Code (Baseline Activity).
- Anti-XSS 4.2 Library.
- CAT.NET, a code analysis tool for .NET.
- Microsoft Source Code Analyzer for SQL Injection.

Phase Four: Verification for LOB

After successful completion of the previous phase, the internal review portion of the Implementation phase, expert application security subject-matter experts are engaged. This phase verifies that an application being deployed into production environments has been developed in a way that adheres to internal security policies and follows industry best practice and internal guidance. Also, another objective is to identify any residual risks not mitigated by application teams.

The assessments conducted during the Verification phase are typically conducted by a security or privacy SME.

Pre-Production Assessment

An ideal comprehensive assessment includes a mix of both white and black box testing. There is a tendency to prefer black box testing because "it's what the hackers do." However, it is also more time consuming and can have mixed results. In addition, it is difficult for individuals who are only "part-time" penetration testers to develop the skills and expertise needed to efficiently perform a black box test. Identifying multiple instances/class of vulnerability bugs is more easily accomplished in a code review (white box). A code review, though, can make finding business logic issues very difficult. Reading the source code for a complex AJAX-based ASP.NET form and actually playing with it can yield vastly different results in terms of issues found.

Further, this phase should be conducted with a mix of manual process and automated tools. Manual reviews may need to be time constrained and focus on high-risk features. Automated tools can reduce overhead, but should not be relied upon exclusively.

Security Requirements

- The service level assigned to the application at the Risk Assessment phase governs the type of assessment an application receives in this phase. An application that has been assigned a medium or higher rating automatically requires a white-box code review, while applications assigned with a low rating will not.
    - **Code review (white box)**
        - Security team is provided access to an application's source code and documentation to aid them in their assessment activities.
        - Complete review using both manual code inspection and security tools, such as static analysis or penetration testing.

- Review threat model. Code reviews are prioritized based on risk ratings identified through threat modeling activities. Components of an application with the highest severity ratings get the highest priority with respect to assigning code review resources, whereas components with low severity ratings are assigned lesser priority.
- Validate tools results. The security expert also validates results from code analysis tools (if applicable), such as CAT.NET to verify that vulnerabilities have been addressed by the development team. In situations where this is not the case, the issue is filed in the bug tracking system.
- If source code is not available or the application is a third-party application, then black box assessment is conducted for that application.
- Code review duration. The duration of a security review is determined by the security SME and is directly related to the amount of code that needs to be reviewed.
- Code review can be conducted manually or by using automated tools to identity categories of vulnerabilities in the code. However, it should be noted that automated tools should supplement a code review and not replace them entirely, due to their limitations.
    - SQL injection. Ensure that the SQL queries are parameterized (preferably within a stored procedure) and that any input used in a SQL query is validated.
    - Cross-site scripting. Ensure that user controlled data is encoded properly before rendering to the browser. .NET applications can leverage Anti-XSS library for encoding data that is more rigorous than the native .NET encoding.
    - Cross-site request forgery. Ensure that the **Page.ViewStateUserKey** property is set to a unique value that prevents one-click attacks on your application from malicious users.
    - Data access. Look for improper storage of database connection strings and proper use of authentication to the database.
    - Input/data validation. Look for client-side validation that is not backed by server-side validation, poor validation techniques, and reliance on file names or other insecure mechanisms to make security decisions.
    - Authentication. Look for weak passwords, clear-text credentials, overly long sessions, and other common authentication problems.
    - Authorization. Look for failure to limit database access, inadequate separation of privileges, and other common authorization problems.
    - Sensitive data. Look for mismanagement of sensitive data by disclosing secrets in error messages, code, memory, files, or the network.
    - Auditing and logging. Ensure the application is generating logs for sensitive actions and has a process in place for auditing logs file periodically.

- Unsafe code. Pay particularly close attention to any code compiled with the **/unsafe** switch. This code does not have all of the protection that normal managed code has. Look for potential buffer overflows, array out of bound errors, integer underflow and overflow, and data truncation errors.
- Unmanaged code. In addition to the checks performed for unsafe code, also scan unmanaged code for the use of potentially dangerous APIs, such as strcpy and strcat. For a list of potentially dangerous APIs, see the section "Potentially Dangerous Unmanaged APIs," in Security Question List: Managed Code (.NET Framework 2.0). Be sure to review any interop calls and the unmanaged code itself to make sure that bad assumptions are not made as execution control passes from managed to unmanaged code.
- Hard-coded secrets. Look for hard-coded secrets in code by looking for variable names, such as "key," "password," "pwd," "secret," "hash," and "salt."
- Poor error handling. Look for functions with missing error handlers or empty catch blocks.
- Web.config. Examine your configuration management settings in the web.config file to make sure that forms authentication tickets are protected adequately, tracking and debugging is turned off, and that the correct algorithms are specified in the machineKey element.
- Code access security. Search for the use of asserts, link demands, and allowPartiallyTrustedCallersAttribute (APTCA).
- Code that uses cryptography. Check for failure to clear secrets and improper use of the cryptography APIs themselves.
- Threading problems. Check for race conditions and deadlocks, especially in static methods and constructors.

- **Penetration test (black box)**
  - This is a flip of a white-box code where the assessment is carried out without access to the application's source code. This testing is intended to simulate an attacker's perspective and uses a combination of tools and penetration techniques to find vulnerabilities in the system.
  - While this best simulates most malicious hacker scenarios, this approach typically yields the least bugs, both in terms of quality and quantity, but it is the best approach when source code is not available for review.
  - Depending upon available resources, this testing can be done internally by your security team or by engaging a third-party security firm as appropriate. Third-party security tools can also help with this requirement. Following are some of the high-level areas to consider in web penetration testing:
    - Use HTTP(s) interrogators, such as Fiddler, to capture traffic and to investigate cookies, headers, and hidden fields. Use Request/Response tampering methods to detect error

disclosure, cross-site scripting, SQL injection, and other injection attacks. All user-controlled data, such as cookies, headers, form fields, and query strings should be tested by sending in malformed data.

- Check for forceful browsing to verify authorization controls in applications where there are more than two user groups with different access levels.
- Use Network Monitor to identify if sensitive data is being transferred from client to server and to verify if the channel is encrypted or not. This would be more useful in the case of thick client LOB applications.
- Experiment with the high risk portions of the application to ensure that controls described in the code review discussion have been implemented correctly and consistently.

- **Deployment review of servers**
  - Review the deployment of the production servers to ensure adequate hardening. This review focuses on minimizing the attack surface of the server (in terms of running services and applications installed), hardening the operating system (ACLs, accounts, patching, registry hardening, minimal open ports, installing server functionality, such as IIS websites on a non-system drive), and hardening functionality, such as IIS and SQL Server.
  - Review, if possible, actual production servers or standard images used to build those servers. Failing that, reviewing test servers with the expectation that the issues found are used as a road map by operations to harden the actual production servers.
  - The test server environment and the production server should have similar security measures while the team is developing the application. This ensures that the application is not modified to execute on the production server. The security team runs security checks on the server. The application security team can complete this either manually or by using a tool.
- **Privacy review**
  - Review the privacy statements, notification, privacy controls, user categories, data management, and PII management for the application.
- **Host security deployment review**
  - Installing server software (IIS, SQL Server) introduces new attack surfaces that must be hardened as well. Each server deployed, whether Intranet- or extranet-facing, needs to be hardened to both reduce attack surfaces and provide defense in depth. Recommended tools: Attack Surface Analyzer

Security Recommendations

- Assessment results yielding Critical or Important bugs automatically result in the application being blocked from deploying into production environments until the issues have been addressed or an exception has been granted by the business owner accepting the risk.
- The security bug bar for LOB applications has additional considerations than what is described earlier in this document. Your business needs to establish guidelines for evaluating the risk posed by individual vulnerabilities. This includes a risk rating framework that applies across all applications. The risk rating framework is independent of the risk assigned to the entire application. The sample table below presents a bug bar that accounts for the unique environment of an LOB application, including the risk posed by individual bugs.

Severity	Description
Critical	• Impact across the enterprise and not just the local LOB application/resources • Exploitable vulnerability in deployed production application
Important	• Exploitable security issue • Policy or standards violation • Affects local application or resources only • Risk rating = High risk
Moderate	• Difficult to exploit • Non-exploitable due to other mitigation • Risk rating = Medium risk
Low	• Bad practice • Non-exploitable • Should not lead to exploit but helpful to attacker exploiting another vulnerability • Risk rating = minimal risk

- There is a trade-off in proving that a vulnerability is actually exploitable against time constraints in finding bugs. It may not be worthwhile to actually craft explicit exploit/malicious payload. In this case, you can adjust the severity as appropriate, erring on the side of caution.

Compliance

Identified risks are logged in the bug-tracking system and assigned a severity rating. The output of this phase results in the following:

- Bug reports
- Exception requests for the risk posed by issues that cannot or will not be fixed prior to production

Handling risk. Development teams may file to be exempt from mitigating such identified risks; however, the important thing to note is that an approved exception request does not relieve development teams of the responsibility to mitigate identified risks indefinitely. Rather, an approved exception request grants development teams a time extension with which risks can exist in production environments unmitigated.

In response to exception requests, security teams gather all pertinent data points, such as technical details, business impact description, interim mitigation, and other exception information, and provide a development team's upper management with these details in the form of an exception form. Upper management can then approve the exception request and accept identified risks for a small period of time, or they reject the exception request and require the business group to mitigate the identified risks. It is important that a specific business owner explicitly assume the risk posed by unmitigated Critical and Important bugs.

Security team tracks all approved exceptions and follows up with the application team after the exception period has expired.

Note: Critical and Important bugs may exist due to a technological or infrastructure limitation that cannot be mitigated in the current release. An exception should be created in to track the issue until such time as the limitation no longer exists.

Resources

- Code review information is available at http://msdn.microsoft.com/en-us/library/ms998364.aspx.
- Security tools:
  - Web debugging proxy tools, such as Fiddler, allow you to inspect all HTTP(S) traffic, set breakpoints and "tamper" with incoming or outgoing data, build custom requests, and replay recorded requests.
  - HTTP passive analysis tools capable of identifying issues related to user-controlled payloads (potential XSS), insecure cookies, and HTTP headers.
  - Microsoft Network Monitor or similar tools that allow you to capture and perform a protocol analysis of network traffic.
  - Browser plug-ins or standalone tools that allow lightweight tampering before data is placed on the wire are also very useful for web security testing.
  - Automated penetration testing tools that crawl publically exposed interfaces (for example, user interfaces and web services) probing for known/common classes of vulnerabilities and known/published exploits.
  - Automated static code analysis tools that parse the syntax of your source code to identify suspected and known vulnerabilities, such as cross-site scripting and code injection.
- Deployment Review Index.
- Privacy Guidelines for Developing Software Products and Services.

Phase Five: Release for LOB

After deployment, several activities need to occur, including regular verification of patch management, compliance, network and host scanning, and responding to any incremental releases for hotfixes and service packs. For the SDL-LOB, these tasks are associated with the post-production assessment.

Post-Production Assessment

The post-production assessment is conducted by the operations team, and the service level is not dictated by the risk level. All applications/hosts/network devices are in scope for assessment on a regular basis. That is, for most organizations, these tasks take place continuously and have existing management processes in place. In which case the application "plugs-in" to those existing processes that monitor changes to the organizational infrastructure rather than occurring as a discrete post-production assessment.

Security Requirements

- The actual list of servers deployed in production will likely vary dramatically from what was initially recorded in the application portfolio at the beginning of the SDL-LOB process. Post-production, operations may own both the servers and routine scanning of those servers for vulnerabilities, patch management, and similar activities. It is a best practice to segregate the duties between the server owners and the compliance organization. The compliance organization owns scanning in a timely manner, and the application team follows the processes established by the compliance team for moving into production.
- Host-level security. Providing security for the host computer involves the following items that are audited on a regular basis on production servers:
    - **Patch management.** The security SME verifies that servers have the latest applicable security updates, including updates from every software manufacturer that has software running on the server.
    - **Appropriate configuration.** The servers are reviewed for compliance with established baselines. For example, all unused services that are not required for the application are disabled and blocked instead of running with default settings.
    - **Antivirus.** Servers have antivirus software running and actively scanning all system file areas, in addition to all shared directories. All systems must have their antivirus application or signature files examined at logon to ensure that the latest antivirus application or current virus signature files are present.

- **Compliance.** Verify compliance with internal business policies and external legal requirements, in addition to standards such as PCI.

- Review access control/permissions. The access control list (ACL) permission settings on all file shares and other system, database, and COM+ objects are reviewed to help prevent unauthorized access. Regular review, for example, of administrator privileges on a given server should be performed.
- Server auditing and logging. Ensuring that auditing with appropriate logging procedures for all system objects that contain business-sensitive information is enabled. Logging procedures include collecting log files and protecting access to log data to only appropriate users (members of security, internal audit, or systems management teams) with the appropriate ACLs. Even more critical is ensuring that the logs are reviewed on a regular basis and that there is some guidance for filtering critical logs from regular operational "noise."
- Network level security. The network infrastructure should be scanned for compliance with baselines (just like servers). This evaluates configuration, vulnerabilities, patch management, and other similar concerns.
- Application retirement. At some point the application will need to be retired gracefully. Are there adequate controls, contact information, and operational awareness to ensure that this can happen at the appropriate time?

Security Recommendations

- Vulnerabilities identified in production should be remediated per operational processes defined by the compliance team.
- Frequently, application teams have a variety of post-production changes to the application, ranging from a hotfix, service pack, or entirely new features. Depending on the scope, the application team either needs to start over by updating the application portfolio (which kicks off a new iteration of the SDL-LOB life cycle), or perform a subset of the SDL-LOB tasks. At a minimum, this subset should include a review/update of the threat model and selected tasks from the **Internal Review** conducting during the Implementation phase.

Resources

- Microsoft Baseline Security Analyzer.
- Microsoft Operation Framework Deliver Phase provides guidance for getting operational concerns reflected during the Requirements phase of project development as well as getting release readiness in place as a validation step prior to production.
- Governance, Risk, and Compliance Service Management.

Appendix A: Privacy at a Glance

This sample document provides basic criteria to consider when building privacy into software releases. It is not exhaustive and should not be treated as such. For more comprehensive guidance, see Privacy Guidelines for Developing Software Products and Services.

Ten Things You Must Do to Protect Privacy

- **Collect user data only if you have a compelling business and user value proposition.** Collect data only if you can clearly explain the net benefit to the user. If you are hesitant to tell users what you plan to do, don't collect their data.
- **Collect the smallest amount of data for the shortest period of time.** Collect personal data only if you absolutely must, and delete it as soon as possible. If there exists a need to retain personal data, ensure that there is business justification for the added cost and risk. Do not collect data for undefined future use.
- **Collect the least sensitive form of data.** If you must collect data, collect it anonymously, if possible. Collect personal data only if you are absolutely certain you need it. If you must include an ID, use one that has a short life span (for example, lasting a single session). Use less sensitive forms of data (for example, telephone area code rather than full phone number). Whenever possible, aggregate personal data from many individuals.
- **Provide a prominent notice and obtain explicit consent before transferring personal data from the user's computer.** Before you transfer any personal data, you must tell the user what data will be transferred, how it will be used, and who will have access to it. Important aspects of the transfer must be visible to the user in the user interface.
- **Prevent unauthorized access to personal data.** If you store or transfer personal data, you must help protect it from unauthorized access, including blocking access to other users on the same system, using technologies that help protect data sent over the Internet, and limiting access to stored data.
- **Get parental consent before collecting and transferring a child's personal data.** Special rules for interacting with children apply any time you know the user is a child (because you know the child's age) or when the content is targeted at or attractive to a child.
- **Provide administrators with a way to prevent transfers.** In an organization, the administrator must have the authority to say whether any data is transferred outside the organization's firewall. You must identify or provide a mechanism that allows the administrator to suppress such transfers. This control must supersede any user preferences.
- **Honor the terms that were in place when the data was originally collected.** If your team decides to use data, its use must be subject to the disclosure terms that were presented to the user when it was collected.

- **Provide users access to their stored personal data.** Users have a right to inspect the personal data you collect from them and to correct it if it is inaccurate—especially contact information and preferences. You also need to ensure that the user is authenticated before they are allowed to inspect or change the information.
- **Respond promptly to user questions about privacy.** Inevitably, some users will have questions about your practices. It is essential that you respond quickly to such concerns. Unanswered questions cause a loss of trust. Be sure a member of your staff is ready to respond whenever a user asks about a privacy issue.

Appendix B: Security Definitions for Vulnerability Work Item Tracking

It is critical for project teams to specify and maintain a work item tracking system that allows for creation, triage, assignment, tracking, remediation, and reporting of software vulnerabilities. Optimally, work item tracking should also include the ability to track security and privacy issues by cause and effect of the security bugs. The work item tracking system should have access controls in place to ensure that changes to information in the system (whether malicious or accidental) can be tracked appropriately.

Ensure that the vulnerability/work item tracking system used includes fields with the following values (at a minimum):

Security Bug Cause

The following fields describe causes of vulnerabilities:

- **Not a Security Bug.** This field is self-explanatory.
- **Buffer Overflow/Underflow.** A failure to check or to limit input data buffer sizes before data is manipulated or processed.
- **Arithmetic Error.** A failure to check bounds conditions for integer math, in which results of calculations might overflow or underflow data type. An example is integer overflow.
- **SQL/Script Injection.** Allows attackers to alter intended behavior by altering script.
- **Directory Traversal.** Allows attackers access to navigate host directory structure.
- **Race Condition.** A security vulnerability caused by code timing or synchronization issues.
- **Cross-Site Scripting.** This cause involves website weaknesses that allow attackers to have inappropriate access to information or resources. Although a subset of script injection, it is listed separately.
- **Cryptographic Weakness.** Insufficient or incorrect use of cryptography to protect data.
- **Weak Authentication.** Insufficient checks or tests to validate that the user or process is who or what it claims to be.
- **Weak Authorization/Inappropriate Permission or ACL.** Access to resources or data for an authenticated user that are not appropriate for users of that type. For example, allowing anonymous or guest users access to sensitive information.
- **Ineffective Secret Hiding.** Insufficient or incorrect protection of cryptographic keys or passwords. For example, storing passwords in plain text in registry or not zeroing out password buffers.

- **Unlimited Resource Consumption (DoS).** A failure to check or limit resource allocations that might allow an attacker to deny service by depleting available resources.
- **Incorrect/No Error Messages.** Insufficient or incorrect reporting of error checking.
- **Incorrect/No Pathname Canonicalization.** An incorrect trust decision based on a resource name, or allowing access to a resource because an attacker bypassed location or name restrictions.
- **Other.** None of the above.

Security Bug Effect

The following definitions are from *Writing Secure Code, Second Edition*.

- **Not a Security Bug.** This field is self-explanatory.
- **Spoofing.** Spoofing threats allow an attacker to pose as another user, allow a rogue server to pose as a valid server, or rogue code to pose as valid code.
- **Tampering.** Data tampering means malicious modification of data.
- **Repudiation.** Repudiation threats are associated with users who deny having performed an action without other parties having any way to prove otherwise. For example, a user with malicious intent performs an illegal operation on a computer that is unable to trace the prohibited operation.
- **Information Disclosure.** Information disclosure threats involve the exposure of information to individuals who are not supposed to have access to it. For example, such a threat might be a user's ability to read a file to which they did not have access, or an intruder's ability to read data in transit between two computers.
- **Denial of Service.** Denial of Service (DoS) attacks deny or restrict services to valid users.
- **Elevation of Privilege.** In this type of threat, a user increases their permissions level and therefore can perform actions they should not be allowed to perform. Any unprivileged user who gains unauthorized access might have sufficient access to compromise or even destroy the system.
- **Attack Surface Reduction.** This type of threat is not a security bug in the same sense as the other items listed. However, when you attempt to reduce the attack surface, it is valuable to track bugs that describe services and functionality that affect the attack surface. It is important to identify attack surface, even though interfaces that are exposed on the attack surface are technically not vulnerabilities. Such bugs are assigned the *Attack Surface Reduction* designation.

Appendix C: SDL Privacy Questionnaire

This sample document provides some criteria to consider when you build a privacy questionnaire. It is not an exhaustive list and should not be treated as such.

Introduction

The following questions are designed to help you complete the privacy aspects of the Security Development Lifecycle (SDL). You will complete some sections, such as the initial assessment and a detailed analysis, on your own. You should complete other sections, such as the privacy review, together with your privacy advisor.

Identify Your Project and Key Privacy Contacts

- What is the name of your project?

  _____

- When will the public first have access to this project?

  _____

- Who on your team is responsible for privacy?

  _____

Initial Assessment

The initial assessment is a quick way to determine your *Privacy Impact Rating* and to estimate the work required to be compliant. The rating (P1, P2, or P3) represents the degree of risk your software presents from a privacy perspective. You need to complete only the steps that apply to your rating. For more detail, see the main Microsoft Security Development Lifecycle document.

Determine Your Privacy Impact Rating

Check all behaviors that apply to your software. If your software does not exhibit any of the behaviors, check "None of the above." For more information, see the Privacy Guidelines for Developing Software Products and Services.

___ Stores personally identifiable information (PII) on the user's computer or transfers it from the user's computer (P1)

___ Provides an experience that targets children or is attractive to children (P1)

___ Continuously monitors the user (P1)

___ Installs new software or changes file type associations, home page, or search page (P1)

___ Transfers anonymous data (P2)

___ None of the above (P3)

Understand Your Obligations and Try to Lower Your Risk (for P1 and P2 Scenarios)

Before you invest time in a design or implementation, get a feel for the work it will take and investigate ways to lower your overall privacy risk. Higher risk translates to higher development and support cost. For more information, see the Privacy Guidelines for Developing Software Products and Services.

Identify a Compliant Design

For more information, see the Privacy Guidelines for Developing Software Products and Services.

Perform a Detailed Privacy Analysis for P1 Scenarios

Before your privacy design review, analyze your threat model to identify any PII that you store or transfer. Summarize the privacy aspects of your software in a detailed analysis.

- Describe the PII you store or data you transfer:

  _____

  _____

- Describe your compelling user value proposition and business justification:

  _____

  _____

- Describe any software you install or changes you make to file types, home page, or search page:

  _____

  _____

- Describe your notice and consent experiences:

  _____

  _____

- Describe how users will access your public disclosure:

  _____

_____

- Describe how organizations can control your feature:

  _____

  _____

- Describe how users can control your feature:

  _____

  _____

- Describe how you will prevent unauthorized access to PII:

  _____

  _____

Conduct a Design Review with Your Privacy Advisor

To avoid costly mistakes, projects with an impact rating of P1 must hold a design review with a privacy advisor before investing heavily in implementation.

Test Your Privacy Experience

Verify that your software complies with privacy requirements. For more information about privacy criteria, see the Privacy Guidelines for Developing Software Products and Services.

Create a Draft Privacy Disclosure

Work with your privacy advisor to write and post a privacy disclosure.

Designate Your Privacy Incident Response Contact

If your software is involved in a privacy incident, your team must be prepared to follow the SDL Privacy Escalation Response Framework (Appendix K in this guide).

Who on your team is the primary contact for Privacy Incident Response?

_____

Obtain Approval from Your Privacy Advisor

Before you ship your project externally, you must obtain approval from your privacy advisor.

Appendix D: Firewall Rules and Requirements

A firewall is a key part of any organization's protection strategy. You should have a consistent policy to manage the settings of your organization's firewall to ensure that users are not exposed to unknown or unnecessary risk from programs that receive unsolicited data over the network. Use the information in this appendix to help draft your organization's firewall policy.

Firewall Rules and Requirements

If you are currently running on Windows XP SP2 or Windows Server 2003, please adhere to the following conditions and requirements.

	Conditions	Requirements
Port-specific rules (permitted *only* when one or more of the following conditions are TRUE)	**1. Customer and User Behavior**	• Empirical data is provided that shows that at least 80% of your users are or will be using a feature that requires the port within the next year.
	2. Informed Consent The user has provided explicit informed consent (the user must be prompted for and grant informed consent for the port to be open).	• Data showing that 80% of customers will answer "Yes" to a dialog asking them whether they want to open the port.
Program-specific rules (permitted *only* when all of the following conditions are true)	1. The program does not run as a service. 2. The program listens on the network only when the user is using the functionality that listens. This includes programs that start when the user logs in. 3. The program can be prevented from starting automatically through a GUI.	

If you are currently running on Windows Vista or Windows Server 2008, please adhere to the following conditions and requirements.

In addition to the port- and program-specific requirements for Windows XP SP2 listed earlier, the following requirements must be met for Windows Vista and Windows Server 2008.

	Requirements
Inbound firewall rules	1. Applications must create rules during setup for traffic that is expected in more than 80% of installations of the application. Explicit user consent is required to enable the rules. 2. Rules must be scoped to all these parameters—Program, Port, and Profile(s). 3. For features that implement services, rules must also be scoped to that service. 4. Services must implement Windows Service Hardening firewall rules.

Application Quality

Programs, applications, services, or other components that wish to receive unsolicited traffic must:

- Produce an independent threat model for the service which identifies each entry point explicitly, including services that are "multiplexed" behind a common port.
- Meet the network fuzzing requirements.

Least Privilege

Firewall rules must adhere to the principle of least privilege by:

- Scoping the rule to "local subnet" or tighter when practical.
- Scoping the rule to only the network profile(s) where the feature is likely to be used. For example, if it is an enterprise feature, then you should scope the rule to domain, private profiles. Unless you expect your feature to be used in a public place like a WiFi hotspot, you should not scope the rule to the public profile.
- Unless your feature requires NAT traversal using transition tunnel technologies, do not set the "Edge" traversal flag.
- Limiting the privileges of the service that use the port to Network Service or more restrictive when practical. When not practical, the threat model should explicitly call out the reasons why.

If services must run with privileges greater than Network Service, it is recommended that the services be split into "privileged" and "non-privileged" components such that only the code that requires higher privileges receives them, and other code is addressed through some IPC mechanism. The end result being that the non-privileged service is the one that receives the traffic.

More information is available at http://www.ece.cmu.edu/%7Edawnsong/papers/privtrans.pdf.

Informed Consent UI

This policy addresses user interface issues, but nothing in the policy should be interpreted to specify a particular user interface. For example, when it says "The user is informed and acknowledges the open port," it does not imply that there must be a dialog that tells the user port 123 has been opened. The

requirement is that the user is informed of the change in some explicit fashion via the UI (not an entry in a log file), and that the details are available for users who want to know.

Terminology

exception
In Windows XP SP2, this is a setting that when enabled allows unsolicited traffic through the firewall. There are two types of exceptions:

- Port exceptions, which allow unsolicited traffic on the specified port.
- Program exceptions, which allow a particular program to receive unsolicited traffic, regardless of port.

The setting may be enabled, which means the traffic is allowed to bypass the firewall, or disabled, which has no effect on the firewall.

In Windows Vista SP1 and Windows Server 2008, this term has been replaced by *inbound firewall rule*.

firewall rule
Firewall rules are created to allow or block a computer sending traffic or receiving traffic over a network. Rules can be created for either inbound traffic or outbound traffic. The rule can be configured to specify traffic that matches specific programs, services, ports, and protocols.

firewall profile
A firewall profile is a way of grouping settings that are applied to the computer depending on the security profile of the network the computer is connected to.

- On Windows Vista and Windows Server 2008, there are three profiles—domain, private, and public.
- On Windows XP SP2, there are two profiles—domain and standard.

domain profile
This profile is applied when the computer is connected to a network in which the computer's domain account resides. Typically, this is the least restrictive profile.

private profile
This profile is applied when the computer is connected to a network like a home or small office network (without an Active Directory infrastructure).

public profile
This profile is applied when the computer is connected to a network in a public place, like a coffee shop or airport.

Windows Service Hardening

Windows Service Hardening restricts services from doing abnormal activities in the file system, registry, network, or other resources that could be used to allow malware to install itself or attack other computers. For example, the Remote Procedure Call (RPC) service can be restricted from replacing system files or modifying the registry.

Appendix E: Required and Recommended Compilers, Tools, and Options for All Platforms

Win32 Requirements: Unmanaged Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C/C++ Compiler	Microsoft Visual Studio .NET 2008		
cl.exe	Version 14.00.50727.42 Use /GS	Use /GS	
Link.exe	Version 8.00.50727.762 Use /SAFESEH Use /NXCOMPAT and don't use /NXCOMPAT:NO. See Appendix F: SDL Requirement: No Executable Pages for more information.	Use /SAFESEH Use /functionpadmin:5 Use /DYNAMICBASE	Visual Studio 2008 SP1 is needed for /DYNAMICBASE
MIDL.exe	Version 6.00.0366 Use /robust	Use /robust	
Source code analysis	Visual Studio 2008 Code Analysis Options ("/analyze") For Visual Studio 2008 code analysis, all warning IDs from the following list must be fixed: 4532 6029 6053 6057 6059 6063 6067 6200 6201 6202 6203 6204 6248 6259 6260 6268 6276 6277 6281 6282 6287 6288 6289 6290 6291 6296 6298 6299 6305 6306 6308 6334 6383	Visual Studio 2008 Code Analysis Options ("/analyze"). For Visual Studio 2008 code analysis, all warning IDs from the following list must be fixed: 4532 6029 6053 6057 6059 6063 6067 6200 6201 6202 6203 6204 6248 6259 6260 6268 6276 6277 6281 6282 6287 6288 6289 6290 6291 6296 6298 6299 6305 6306 6308 6334 6383 Standard Annotation Language	Visual Studio 2008 Team Edition contains a publicly available version that is branded as "C/C++ Code Analysis."

		(SAL): Code annotated with SAL should correct additional warnings, in addition to those listed above. See Appendix H: SDL Standard Annotation Language (SAL) Recommendations for Native Win32 Code for more information. The warnings are summarized as follows: **SAL Compliance** Visual Studio 2008: 26020–26023 **/analyze** Visual Studio 2008: 6029 6053 6057 6059 6063 6067 6201–6202 6248 6260 6276 6277 6305	
Protecting Against Heap Corruption	n/a	All executable programs written using unmanaged code (.EXE) must call the HeapSetInformation interface. See Appendix I: SDL Requirement: Heap Manager Fail Fast Setting for more information.	
C4700 and C4701 Compiler Warnings	n/a	Compile code with C4700 and C4701 compiler warnings enabled and fix all instances of these warnings.	

Win32 Requirements: Managed Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C# Compiler	Visual Studio 2008		If using C#, use C# v2.0 or later; if using

			Visual Basic.NET use Visual Basic.NET 8.0 or later
csc.exe	Version 8.0.50727.42		
.NET Framework	Version 2.0.50727		
FxCop	Version 1.32	Most recent version	

Win32 Requirements: Testing Tools

Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
AppVerifier	Most recent version Run tests as described in Appendix J: SDL Requirement: Application Verifier.	Most recent version	**Note:** AppVerifier is targeted at unmanaged code and is not optimized for managed code.

Win64 Requirements (IA64 and AMD64): Unmanaged Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C/C++ Compiler	Visual Studio 2008		
cl.exe	Version 14.00.50727.42		
Link.exe	Version 8.00.50727.762 Use of /SAFESEH does not apply to Win64 platforms. Use /NXCOMPAT and do not use /NXCOMPAT:NO. See Appendix F: SDL Requirement: No Executable	AMD64 only: Use /functionpadmin:6 Use of /SAFESEH does not apply to Win64 platforms. Use /DYNAMICBASE	Visual Studio 2008 SP1 is needed for /DYNAMICBASE.

	Pages for more information.		
MIDL.exe	Version 6.00.0366 Use /robust	Use /robust	
Protecting Against Heap Corruption	n/a	All executable programs written using unmanaged code (.EXE) must call the HeapSetInformation interface. See Appendix I: SDL Requirement: Heap Manager Fail Fast Setting for more information.	
C4700 and C4701 Compiler Warnings	n/a	Compile code with C4700 and C4701 compiler warnings enabled and fix all instances of these warnings.	

Win64 Requirements (IA64 and AMD64): Managed Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C# Compiler	Visual Studio 2008		If using C#, use C# v2.0 or later; if using Visual Basic.NET use Visual Basic.NET 8.0 or later
csc.exe	Version 8.0.50727.42		
.NET Framework	Version 2.0.50727		
FxCop	Most recent version	Most recent version	

Win64 Requirements (IA64 and AMD64): Testing Tools

Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
AppVerifier	Most recent version	Most recent version	**Note:** AppVerifier is targeted at

	Run tests as described in Appendix J: SDL Requirement: Application Verifier.		unmanaged code and is not optimized for managed code.

Windows CE Requirements: Unmanaged Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C/C++ Compiler	Visual Studio 2008		
cl.exe	Version 14.0.50725.0 Use –GS (see comments)	Use –GS (see comments)	The –GS flag has a modest impact on code size, which can be of interest on WinCE platforms. Minimally, –GS must be used on all Internet-facing code. Ideally, –GS should be used on all code.
Link.exe	Version 8.00.50727.762 Use of /SAFESEH only applies to x86 on WinCE platforms. Use of /NXCOMPAT does not apply to WinCE.	Use of /SAFESEH only applies to x86 on WinCE platforms. Use of /NXCOMPAT:NO does not apply to WinCE.	
MIDL.exe	Version 6.00.0366 Use /robust	Use /robust	
Source code analysis	Visual Studio 2008 Code Analysis Options (“/analyze) For Visual Studio 2008 code analysis, all warning IDs from		

	the following list must be fixed: 4532 6029 6053 6057 6059 6063 6067 6200 6201 6202 6203 6204 6248 6259 6260 6268 6276 6277 6281 6282 6287 6288 6289 6290 6291 6296 6298 6299 6305 6306 6308 6334 6383		

Windows CE Requirements: Compact Framework Managed Code

Compiler/Tool	Minimum Required Version and Switches/Options	Optimal/Recommended Version and Switches/Options	Comments
C# Compiler	Visual Studio 2008		If using C#, use C# v2.0 or later; if using Visual Basic.NET use Visual Basic.NET 8.0 or later
csc.exe	Version 8.0.50727.42		
.NET Framework	Version 2.0.50727		
FxCop	Most recent version	Most recent version	

Appendix F: SDL Requirement: No Executable Pages

Executing code from data code pages is a very common attack vector. The use of this technique is needed only in very limited scenarios. As a result, all programs and services should avoid this technique unless explicitly required. Having even one page marked EXECUTABLE in a process (other than dynamic-link libraries or DLLs) usually renders other security measures (such as /GS and /SafeSEH) useless for that process.

Goals and Justification

The Windows Exception Handling mechanism assumes that it is safe to dispatch exceptions to any address if they are not in a DLL but are still EXECUTABLE.

Given a stack buffer overflow, an attacker can overflow the nearest exception record on the stack (there is always at least one) and point it to an address in the page marked EXECUTABLE.

Because of the number of stack locations controlled by the overflow at the point the exception handler takes over, many possible op-code sequences would reliably deliver execution back to the attack-supplied buffer. One such sequence is {pop, pop, ret} (possibly interleaved with other instructions). It is also possible to leverage a sequence of op-codes that would produce an arbitrary memory-overwrite in a two-stage attack.

Because of the number of possibilities, it is very hard to prove that bytes on a page marked EXECUTABLE cannot be abused sufficiently to take control.

Scope

The following subsections specify the scope of the No Executable Pages requirement proposal.

Operating Systems

This requirement applies to Win32 and Win64 operating systems but not to Windows CE or Macintosh.

Products/Services

This requirement applies to code that runs on users' computers (products) and to code that runs only on Microsoft-owned computers and accessed by users (for example, Microsoft-owned and provisioned online services).

Technologies

This requirement applies to unmanaged (native) code, such as C and C++.

New Code and Legacy Code

This requirement applies to both new code and legacy code.

Exceptions

Sometimes it simply might not be possible to mark all pages as non-executable. Examples include digital rights management technologies and just-in-time (JIT) technologies that dynamically create code. For such cases, the following techniques can help make the pages safe:

- If your code uses VirtualAllocXXX APIs to allocate EXECUTABLE pages, load a dummy DLL and use one of its sections instead.
- Register a Vectored Exception Handler in the process, and vet the chain of exception handlers to make sure none of them point to your EXECUTABLE pages.
- Randomize the address of the EXECUTABLE pages and/or randomize the starting offset of content within those pages.

Special Cases

Because marking a binary as "DEP compatible" (with /NXCOMPAT) changes how the operating system interacts with the executable, it is important to test all executables (.EXE) marked as /NXCOMPAT with a version of Windows that supports this functionality:

- Client software: Windows XP SP2, Windows Vista
- Server software: Windows Server 2003 Service Pack 1 (SP1) or Windows Server 2008

(Review the detailed description of the Data Execution Prevention [DEP] feature for specific details about how to use DEP on Windows Server 2003 SP1.)

All executables (.EXE) marked as /NXCOMPAT are able to take advantage of Data Execution Protection. Dynamic-link libraries (.DLL files) or other code called by executables (such as COM objects) do not gain direct security benefits with /NXCOMPAT but need to coordinate enabling /DEP support with any executable files that might call them. Any EXE with /NXCOMPAT enabled that loads other code without /NXCOMPAT enabled may have the process fail unexpectedly unless the EXE and all of the other code that it calls (such as DLLs or COM objects) have been thoroughly tested with DEP enabled (linked with /NXCOMPAT option).

Requirements

Requirement: Do Not Use Certain VirtualAllocXXX Flags

If your code calls any of these APIs:

- VirtualAlloc
- VirtualAllocEx
- VirtualProtect
- VirtualProtectEx

- NtAllocateVirtualMemory
- NtProtectVirtualMemory

Do not use any of these flags:

- PAGE_EXECUTE
- PAGE_EXECUTE_READ
- PAGE_EXECUTE_READWRITE
- PAGE_EXECUTE_WRITECOPY

Requirement: Use /NXCOMPAT Linker Option

All binaries must link with /NXCOMPAT flag (and not link with /NXCOMPAT:NO) using the linker included with Visual Studio 2005 and later.

Compliance Measurement

Requirement: Do Not Use Certain VirtualAllocXXX Flags

While no solutions exist to monitor the use of these APIs, project teams will be asked to examine project specifications for the use of these APIs and attest to their removal.

Requirement: Use /NXCOMPAT Linker Option

Requirement: Measurement by Product/Service Team

Project teams will be asked to attest to the use of this linker option.

Requirement: Measurement by Security Advisors

This requirement is required to complete the final security review, and use of this linker option should be confirmed before allowing the software to be released to manufacturing or the web (RTM/RTW).

Support Considerations

Requirement: Impact on Existing SDL Requirements

This requirement currently has some overlap with the Banned API requirement in that it describes some function calls that are prohibited. With the release of Windows Vista, this requirement is an even more important part of the Microsoft defense-in-depth strategy (in conjunction with Address Space Layout Randomization [ASLR] support in Windows Vista).

Requirement: Education and Training

Education reference materials currently available include:

- Data Execution Prevention
- Detailed Description of the Data Execution Prevention (DEP) Feature

Appendix G: SDL Requirement: No Shared Sections

Binaries that are shipped as part of the product must not contain sections marked as *shared*, which are a security threat and should not be used. Use properly secured, dynamically created shared memory objects instead.

Rationale

The Portable Executable (PE) format allows binaries to define *sections*—named areas of code or data—that have distinct properties, such as size, virtual address, and flags, which define the behavior of the operating system as it maps the sections into memory when the binary image is loaded. An example of a section would be *text*, which is typically present in all executable images. This section is used to store the executing code and is marked as *Code, Execute, Read,* which means code can execute from it, but data cannot be written to it. It is possible to define custom sections with desired names and properties by using compiler/linker directives.

One such section flag is *Shared*. When it is used and the binary is loaded into multiple processes, the shared section maps to the same physical memory address range. This functionality makes it possible for multiple processes to write to and read from addresses that belong to the shared section.

Unfortunately, it is not possible to secure a shared section. Any malicious application that runs in the same session can load the binary with a shared section and eavesdrop or inject data into shared memory (depending on whether the section is read-only or read-write).

To avoid security vulnerabilities, use the **CreateFileMapping** function with proper security attributes to create shared memory objects.

Detecting Existing Shared PE Sections

You can use the following linker directive to create a shared section:

```
/section:<name>, RWS
```

The following directives in C/C++ source code can be used:

```
// (this introduces a new PE section)
#pragma data_seg(".shared")
            int mySharedData = 1;
#pragma data_seg()
```

Or:

```
#pragma section(".shrd2", read, write, shared)
__declspec(allocate(".shrd2")) int mySharedData2 = 2;
```

Resources

- Creating Named Shared Memory

Appendix H: SDL SAL Recommendations for Native Win32 Code

The Standard Source Code Annotation Language (SAL), a technology from Microsoft Research that is actively embraced by the Windows and Office teams, is a powerful addition to C/C++ source code to help find bugs, especially security vulnerabilities. SAL can help find more vulnerabilities than the present set of static analysis tools can find. A major benefit of SAL is that developers need only annotate their headers to provide benefit for others. For example, most C runtime and Windows headers that ship with Visual Studio 2005 and later are annotated. Windows Development Kit headers are also annotated.

All products developed using SDL should use a subset of SAL to help find deeper issues, such as buffer overrun issues. Microsoft teams, such as Office and Windows, are using SAL beyond the requirements of SDL.

SAL Details

SAL is primarily used as a method to help tools, such as the Visual C++ /analyze compiler option, to find vulnerabilities by knowing more about a function interface. For the purposes of this appendix, SAL can document three properties of a function:

- Whether a pointer can be NULL
- How much space can be written to a buffer
- How much can be read from a buffer (potentially including NULL termination)

At a high level, things become more complicated because there are two implementations of SAL:

- __declspec syntax (Visual Studio 2005 and Visual Studio 2008)
- Attribute syntax (Visual Studio 2008)

Each of these implementations maps onto lower-level primitives that are too verbose for typical use. Therefore, developers should use macros that define commonly used combinations in a more concise form. C or C++ should add the following to their precompiled headers:

- #include “sal.h”

SAL Recommendations

- Start with new code only. Microsoft strongly recommends that you also plan to annotate old code.
- All function prototypes that accept buffers in internal header files you create should be SAL annotated.
- If you create public headers, you should annotate all function prototypes that read or write to buffers.

SAL in Practice

All examples use the __declspec form.

A classic example is that of a function that takes a buffer and a buffer size as arguments. You know that the two arguments are closely connected, but the compiler and the source code analysis tools do not know that. SAL helps bridge that gap.

A list of common SAL annotations, with examples, can be found on Michael Howard's blog.

The following code demonstrates this benefit by annotating a writeable buffer named buf:

```
void FillString(
  TCHAR* buf,
  int cchBuf,
  TCHAR ch) {
  for (int i = 0; i < cchBuf; i++)
    buf[i] = ch;
}
```

cchBuf is the character count of buf. Adding SAL helps link the two arguments together:

```
void FillString(
  __out_ecount(cchBuf) TCHAR* buf,
  int cchBuf,
  TCHAR ch) {
  for (int i = 0; i < cchBuf; i++)
    buf[i] = ch;
}
```

If you compile this code for Unicode, a potential buffer overrun exists when you call this code:

```
TCHAR buf[MAX_PATH];
FillString(buf, sizeof(buf), '\0');
```

sizeof is a byte count, not a character count. The programmer should have used countof.

In the __out_ecount macro, __out means the buffer is an "out" buffer and is written to by the functions. The buffer size, in elements, is _ecount(cchBuf). Note that this function cannot handle a NULL buf, if it could, then the following macro could be used: __out_ecount_opt(cchBuf), where _opt means optional.

The following example shows a function that reads to a buffer and writes to another.

```
void CopyRange(__in_ecount(cchFrom) const char *from,
      size_t cchFrom,
      __out_ecount(cchTo) char *to,
      size_t cchTo);
```

Tools Usage

To take advantage of SAL, make sure you compile your code with a version of Microsoft Visual C++® 2005 or Visual C++ 2008 that support the /analyze compile-time flag.

Top Severity Warnings to Triage for Fixing

6029 Possible buffer overrun in call to <function>: use of unchecked value

6053 Call to <function>: may not zero-terminate string <variable>

6057 Buffer overrun due to number of characters/number of bytes mismatch in call to <function>

6059 Incorrect length parameter in call to <function>: pass the number of remaining characters, not the buffer size of <variable>

6200 Index <name> is out of valid index range <min> to <max> for non-stack buffer <variable>

6201 Buffer overrun for <variable>, which is possibly stack allocated: index <name> is out of valid index range <min> to <max>

6202 Buffer overrun for <variable>, which is possibly stack allocated, in call to <function>: length <size> exceeds buffer size <max>

6203 Buffer overrun for buffer <variable> in call to <function>: length <size> exceeds buffer size

6204 Possible buffer overrun in call to <function>: use of unchecked parameter <variable>

6209 Using "sizeof<variable1>" as parameter <number> in call to <function> where <variable2> may be an array of wide characters; did you intend to use character count rather than byte count?

6248 Setting a SECURITY_DESCRIPTOR's DACL to NULL will result in an unprotected object

6383 Buffer overrun due to conversion of an element count into a byte count

Benefits of SAL

Because SAL provides more function interface information to the compiler toolset, SAL finds more issues earlier and with less noise.

Resources

More detailed SAL information can be found in Chapter 1 of *Writing Secure Code for Windows Vista* from Howard and LeBlanc and Howard's blog.

Summary

- Microsoft recommends that you start by annotating new code only. As time permits, existing code should be annotated also.
- You should use SAL for all functions that write to buffers.
- You should consider using SAL for all functions that read from buffers.
- The SDL requirement does not mandate either SAL macro syntax. Use attribute or __declspec as you see fit.
- Annotate the function prototypes in headers that you create.
- If you consume public headers, you must use only annotated headers.

Appendix I: SDL Requirement: Heap Manager Fail Fast Setting

During the past few years, Microsoft added core defenses at the operating system level to help protect against some types of attacks. None of them are perfect but, when used together, they can provide an effective defense. Examples include the firewall, the –GS flag, heap checking, and DEP (also known as *NX*). Generally, Microsoft introduces an initial version or subset of the defense and then augments it over time, as developers and end users become accustomed to it.

A good example is Data Execution Protection (DEP). This feature was never enabled in Microsoft Windows 2000 because it had no hardware support but became available in Windows Server 2003 as an unsupported boot.ini option. DEP was then supported for the first time in Windows XP SP2 but set only for the system and not for non-system applications.

Another example, and the focus of this requirement, is the ability to detect and respond to heap corruption. In the past, there was no protection in the heap from heap-based buffer overruns. Microsoft then added metadata checking, primarily in the form of forward and backward link-checking post block-free to determine whether a heap overrun had occurred. However, for application compatibility reasons, the mitigation was limited to preventing the arbitrary write controlled by a potential exploit from taking place, and the application was allowed to continue to run after the point the corruption was detected. Windows Vista includes a more robust mechanism—the application terminates when heap corruption is detected. This mechanism also helps developers find and fix heap-based overruns early in the development lifecycle.

This Heap Manager Fail Fast Setting requirement might cause reliability issues in applications that have poor heap memory management. However, the failing code is found immediately and can be fixed, which makes software both more secure and more reliable. Windows Vista has encountered only one such example in a third-party ActiveX control.

This capability is enabled for some core operating system components but not for non-system applications running on Windows Vista. This appendix outlines how to enable the option for non-system applications.

Goals and Justification

Currently, even if corruption is detected in the heap manager, the process might continue to run successfully, depending on the corruption pattern, the data affected, and the usage. Some applications might hide memory-related vulnerabilities by handling exceptions, such as access violations, that are raised inside the heap manager. The goal is to find vulnerabilities early and to create robust code.

Therefore, certain critical operating system components must hard fail when heap corruption is detected. Also, any new application developed and tested on Windows Vista should terminate on heap corruptions.

This requirement has two major benefits:

- The first benefit applies to development. With this requirement in place, problematic code is more likely to be found because the failure is immediate. Think of it as an "assert" on heap overrun. However, the code that performs heap-based memory allocation and manipulation must be tested correctly. The best method to find this class of vulnerability is through fuzz testing.
- The second benefit is in deployment. If a vulnerability is missed during development and a heap-based overrun exploit occurs, the exploit would become a denial-of-service issue rather than a potential code execution issue.

The requirement is a no-op on versions of the Windows operating system prior to Windows Vista because the required setting is ignored.

Scope

The following subsections specify the scope of the Heap Manager Fail Fast Setting requirement proposal.

Operating System

This requirement applies only to Win32.

Products/Services

This proposal applies to code that runs on users' computers (products) and to code that runs only on Microsoft-owned computers and accessed by users (services).

Technologies

This proposal applies to unmanaged (native) code, such C and C++, but not to managed code, such as C#.

New Code and Legacy Code

This proposal applies to new code but not to legacy code.

External Applicability

This proposal applies to external third-party ISV code.

Exceptions and Special Cases

There are no exceptions or special cases for new code.

Requirement Definition

Before you use any heap–based memory, you must add the following code to your application startup:

```
(void)HeapSetInformation(NULL,
HeapEnableTerminationOnCorruption,
NULL,
0);
```

Microsoft also recommends that the code use the Low-Fragmentation Heap, which has been shown to be more resistant to attack than the "normal" heap in Windows. To use the Low-Fragmentation Heap, use the following code:

```
DWORD Frag = 2;
(void)HeapSetInformation(NULL,
HeapCompatibilityInformation,
&Frag,
sizeof(&Frag));
```

You must add these function calls as early as possible on application startup. This requirement applies to all unmanaged .EXE files, but not to dynamic-link libraries (DLLs), which do not need to call this function.

Compliance Measurement

A product/service team can verify compliance in either of the two ways mentioned in the following requirement.

Requirement: Measurement by Product/Service Team

- Verify that the correct function is included in the **main()** function of the product, and attest to its use.

  Or

- Run the application under a kernel debugger, issue the **!heap -s** command, and verify that the following text appears: **Termination on corruption : ENABLED**.

Appendix J: SDL Requirement: Application Verifier

Application Verifier is a runtime verification tool for unmanaged code. It helps developers quickly find subtle programming errors that can be extremely difficult to identify with typical application testing. Application Verifier makes it easier to create reliable applications by monitoring an application's interaction with the Windows operating system. It profiles the application's use of kernel objects, the registry, the file system, and Win32 APIs (heap, handle, locks, and more).

Why Is Application Verifier Important?

Application Verifier can help quickly identify security issues related to heap buffer overruns by enabling it when test scenarios are covered. As a result of using it, your organization could avoid having to release security bulletins related to such problems and save you both money and credibility.

Code Required to Run Application Verifier

Application Verifier should be run on all unmanaged code.

Application Verifier is a tool that detects errors in a process (user-mode software) while the process is running. Typical findings include heap corruptions (including heap buffer overruns) and incorrect synchronizations and operations. Whenever Application Verifier finds an issue, it goes into debugger mode. Therefore, either the application being verified should run under a user-mode debugger or the system should run under a kernel debugger.

Application Verifier Usage Scenarios

Application Verifier (available in Visual Studio and as a download) cannot be enabled for a running process. You need to make settings as described in this appendix and then start the application. The settings are persistent until explicitly deleted. Therefore, an application always starts with AppVerifier enabled, regardless of how many times you launch it

The scenarios in this appendix showcase the recommended command-line options for quality gates that you should run during all tests (BVTs, stress, unit, and regression) that exercise the code change.

Testing with Application Verifier

The expectation for this scenario is that the application does not break into debugger mode and that all tests pass with the same pass rate as when run without Application Verifier enabled.

1. Enable verifier for the application(s) you wish to test using:

```
appverif /verify <MyApp.exe>
```

Note: /verify enables the base checks: HANDLE_CHECKS, RPC_CHECKS, COM_CHECKS, LOCK_CHECKS, FIRST_CHANCE_EXCEPTION_CHECKS, and FULL_PAGE_HEAP.

2. If you are testing a dynamic-link library (DLL), you must enable the verifier for the test .exe that is exercising the DLL.
3. Run *all* your tests exercising the application.
4. Analyze any debugger break that you encounter. Debugger breaks signify bugs found by the verifier, and you need to understand and fix them.
5. When you are finished, delete all settings made with:

```
appverif /n <MyApp.exe>
```

You can debug any issues you find with Application Verifier by reviewing the Verifier Stop codes within the help contents.

Testing with Application Verifier and Fault Injection

The expectation for this scenario is that the application does not break into debugger mode. Not breaking into debugger mode means there are no errors that need to be addressed.

The pass rate for the tests may decrease significantly because random fault injections are introduced into the normal operation.

1. Enable verifier and fault injection for the application(s) you wish to test by using the following command-line syntax:

```
appverif /verify <MyApp.exe> /faults
```

Note: If you are testing a DLL, you can apply fault injection on a certain DLL instead of on the entire process. The command-line syntax would be:

```
appverif /verify TARGET [/faults [PROBABILITY [TIMEOUT [DLL …]]]]
```

For example, appverif /verify <*mytest.exe*> /faults 5 1000 d3d9.dll

2. Run *all* your tests exercising the application.
3. Analyze any debugger break that you encounter. Debugger breaks signify bugs found by the verifier, and you need to understand and fix them.
4. When you are finished, delete all settings made with:

```
appverif /n <MyApp.exe>
```

Note that running with and without fault injection exercises different code paths in an application. Therefore, you must run both scenarios to obtain the full benefit of Application Verifier.

Appendix K: SDL Privacy Escalation Response Framework (Sample)

This sample document provides basic criteria to consider when building a privacy breach response process.

Purpose

The purpose of the Privacy Escalation Response Framework (PERF) is to define a systematic process that you can use to resolve privacy escalations efficiently. The process must also manage the associated internal and external communications and identify the root cause or causes of each escalation so that policies or processes can be improved to help prevent recurrences.

Definition: Privacy Escalation

A *privacy escalation* is an internal process to communicate the details of a privacy-related incident. A privacy escalation is warranted for the following types of incidents:

- Data breaches or theft
- Failure to meet communicated privacy commitments
- Privacy-related lawsuits
- Privacy-related regulatory inquiries
- Contact from media outlets or a privacy advocacy group regarding a privacy incident

Privacy Escalation Team

Your privacy escalation core team should include an escalation manager, a legal representative, and a public relations (PR) representative, at a minimum. The escalation manager should be responsible for including appropriate representation from across your organization (such as privacy and business experts) and for driving the process to completion. The legal and public relations representatives are responsible for helping resolve any legal or PR concerns consistently throughout the process.

Submitting Privacy Escalation Requests

The privacy core team should set up a distribution group or managed e-mail account that any employee can contact regarding a potential privacy escalation.

Privacy Escalation Response Process

Escalation should begin when the first e-mail notification of the issue is received. The escalation manager is responsible for evaluating the content of the escalation to determine whether more information is

required. If so, the escalation manager is responsible for working with the reporting party and other contacts to determine:

- The source of the escalation.
- The impact and breadth of the escalation.
- The validity of the incident or situation.
- A summary of the known facts.
- Timeline expectations.
- Employees who know about the situation, product, or service.

The escalation manager should then disseminate this information to appropriate contacts and seek resolution. Although the escalation manager can assign portions of the workload to other people as needed, the escalation manager should ensure that all aspects of the escalation are resolved. Appropriate resolutions should be determined by a privacy escalation core team in cooperation with the reporting party and other applicable contacts. Appropriate resolutions might include some or all of the following:

- Internal incident management
- Communications and training
- Human resources actions, in the case of a deliberate misuse of data
- External communications, such as:
    - Online Help articles
    - Public relations outreach
    - Breach notification
    - Documentation updates
    - Short-term and/or long-term product or service changes

Closing

After all appropriate resolutions are in place, the privacy escalation team should evaluate the effectiveness of privacy escalation response actions. An effective remediation is one that resolves the concerns of the reporting party, resolves associated user concerns, and helps to ensure that similar events do not recur.

Appendix L: Glossary

buffer overflow

A condition that occurs because of a failure to check or to limit input data buffer sizes before data is manipulated or processed.

bug bar

A set of criteria that establishes a minimum level of quality.

deprecation

Designating a component for future removal from a software program.

fuzz testing

A means of testing that causes a software program to consume deliberately malformed data to see how the program reacts.

giblets

Code that was created by external development groups in either source or object form.

harden

Take steps to ensure no weaknesses or vulnerabilities in a software program are exposed.

implicit consent

An implied form of consent in certain limited home and organizational networking scenarios.

informed consent

An explicitly stated form of consent that is usually provided after some form of conditions acknowledgment.

penetration testing (pen testing)

A test method in which the security of a computer program or network is subjected to deliberate simulated attack. See http://en.wikipedia.org/wiki/Penetration_Testing, for additional information.

personally identifiable information (PII)

Data that provides personal or private information that should not be publicly available. Examples include financial or medical information.

port exception

An exception to a firewall policy that specifies a certain logical port in the firewall should be opened or closed.

privacy escalation

An internal process to communicate the details of a privacy-related incident. A privacy escalation is typically warranted for data breaches or theft, failure to meet communicated privacy commitments, privacy-related lawsuits, privacy-related regulatory inquiries, and contact from media outlets or a privacy advocacy group regarding a privacy incident.

privacy impact rating

A measurement of the sensitivity of the data a software program processes from a privacy perspective.

privacy lead or privacy champ

An individual on a software development team who is responsible for privacy for the software program being developed.

program exception

An exception to a firewall policy that exempts a specific program or programs from some aspect of the policy.

security push

A team-wide focus on threat model updates, code review, testing, and documentation scrub. Typically, a security push occurs after a product is code/feature complete.

service pack (SP)

A means by which product updates are distributed. Service packs might contain updates for system reliability, program compatibility, security, or privacy. A service pack requires a previous version of a product before it can be installed and used. A service pack might not always be named as such; some products may refer to a service pack as a *service release*, *update*, or *refresh*.

zero-day exploit

An exploit of a vulnerability for which a security update does not exist.

Appendix M: SDL Privacy Bug Bar (Sample)

Note: This sample document is for illustration purposes only. The content presented below outlines basic criteria to consider when creating privacy processes. It is not an exhaustive list of activities or criteria and should not be treated as such.

Please refer to the definitions of terms in this section.

End-User Scenarios **Usage notes:** These scenarios apply to consumers, enterprise clients, and enterprise administrators acting as end users. For enterprise administrators acting in their administrative role, see the Enterprise Administrators Scenarios.	

Critical	• Lack of notice and consent
	  • Example: Transfer of sensitive personally identifiable information (PII) from the user's system without prominent notice and explicit opt-in consent in the UI prior to transfer.
	• Lack of user controls
	  • Example: Ongoing collection and transfer of non-essential PII without the ability within the UI for the user to stop subsequent collection and transfer.
	• Lack of data protection
	  • Example: PII is collected and stored in a persistent general database without an authentication mechanism for users to access and correct stored PII.
	• Lack of child protection
	  • Example: Age is not collected for a site or service that is attractive to or directed at children and the site collects, uses, or discloses the user's PII.
	• Improper use of cookies
	  • Example: Sensitive PII stored in a cookie is not encrypted.
	• Lack of internal data management and control
	  • Example: Access to PII stored at organization is not restricted only to those who have a valid business need or there is no policy to revoke access after it is no longer required.

	• Insufficient legal controls • Example: Product or feature transmits data to an agent or independent third party that has not signed a legally approved contract.

Important	• Lack of notice and consent
	  • Example: Transfer of non-sensitive PII from the user's computer without prominent notice and explicit opt-in consent in the UI prior to transfer.
	• Lack of user controls
	  • Example: Ongoing collection and transfer of non-essential anonymous data without the ability in the UI for the user to stop subsequent collection and transfer.
	• Lack of data protection
	  • Example: Persistently stored non-sensitive PII lacks a mechanism to prevent unauthorized access. A mechanism is not required where the user is notified in the UI that data will be shared (for example, folder labeled "Shared").
	• Data minimization
	  • Example: Sensitive PII transmitted to an independent third party is not necessary to achieve the disclosed business purpose.
	• Improper use of cookies
	  • Example: Non-sensitive PII stored in a persistent cookie is not encrypted.

Moderate	• Lack of user controls
	  • Example: PII is collected and stored locally as hidden metadata without any means for a user to remove the metadata. PII is accessible by others or may be transmitted if files or folders are shared.
	• Lack of data protection
	  • Example: Temporarily stored non-sensitive PII lacks a mechanism to prevent unauthorized access during transfer or storage. A mechanism is not required where the sharing of information is obvious (for example, user name) or there is prominent notice.
	• Data minimization
	  • Example: Non-sensitive PII or anonymous data transmitted to an independent third party is not necessary to achieve disclosed business purpose.
	• Improper use of cookies
	  • Example: Use of persistent cookie where a session cookie would satisfy the

	purpose. Or, persisting a cookie for a period that is longer than necessary to satisfy the purpose. • Lack of internal data management and control • Example: Data stored at organization does not have a retention policy.
Low	• Lack of notice and consent • Example: PII is collected and stored locally as hidden metadata without discoverable notice. PII is not accessible by others and is not transmitted if files or folders are shared.

Enterprise Administration Scenarios **Usage notes:** These scenarios apply to enterprise administrators acting in their administrative role. For Enterprise administrators in an end-user role, see the End User Scenarios.	

Critical	• Lack of enterprise controls
	  • Example: Automated data transfer of sensitive PII from the user's system without prominent notice and explicit opt-in consent in the UI from the enterprise administrator prior to transfer.
	• Insufficient privacy disclosure
	  • Example: Deployment or development guide for enterprise administrators provides legal advice.

Important	• Lack of enterprise controls
	  • Example: Automated data transfer of non-sensitive PII or anonymous data from the user's system without prominent notice and explicit opt-in consent in the UI from the enterprise administrators prior to transfer. Notice and consent must appear in the UI—not through the End-User License Agreement (EULA) or Terms of Service.
	• Insufficient privacy disclosure
	  • Example: Disclosure to enterprise administrators, such as deployment guide or UX, does not disclose storage or transfer of PII.

Moderate	• Lack of enterprise controls • Example: No mechanism is provided or identified to help the enterprise administrators prevent accidental disclosure of user data (for example, set site permissions).

Definition of Terms

anonymous data
Non-personal data that has no connection to an individual. By itself, it has no intrinsic link to an individual user. For example, hair color or height (in the absence of other correlating information) does not identify a user.

child or children
Under 14 years of age in Korea and under 13 years of age in the United States.

discoverable notice
A discoverable notice is one the user has to find (for example, by locating and reading a privacy statement of a website or by selecting a privacy statement link from a Help menu).

discrete transfer
Data transfer is discrete when it is an isolated data capture event that is not ongoing.

essential metadata
Metadata that is necessary to the application for supporting the file (for example, file extension).

explicit consent
Explicit consent requires that the user take—or have the ability to take—an explicit action before data is collected or transferred.

hidden metadata
Hidden metadata is information that is stored with a file but is not visible to the user in all views. Hidden data may include personal information or information that the user would likely not want to distribute publicly. If such information is included, the user must be made aware that this information exists and must be given appropriate control over sharing it.

implicit consent
Implicit consent does not require an explicit action indicating consent from the user; the consent is implicit in the operation the user initiates.

non-essential metadata
Metadata that is not necessary to the application for supporting the file (for example, key words).

persistent storage
Persistent storage of data means that the data continues to be available after the user exits the application.

personally identifiable information (PII)
Personally identifiable information is any information (i) that identifies or can be used to identify, contact, or locate the person to whom such information pertains, or (ii) from which identification or contact information of an individual person can be derived. Personally Identifiable Information includes, but is not limited to, name, address, phone number, fax number, e-mail address, financial profiles, medical profile, social security number, and credit card information. Additionally, to the extent that unique information (which by itself is not PII, such as a unique identifier or IP address) is associated with PII, such unique information will also be considered PII.

prominent notice
A prominent notice is one that is designed to catch the user's attention. Prominent notices should contain a high-level, substantive summary of the privacy-impacting aspects of the feature, such as what data is being collected and how that data will be used. The summary should be fully visible to a user without additional action on the part of the user, such as having to scroll down the page. Prominent notices should also include clear instructions for where the user can get additional information (such as in a privacy statement).

sensitive PII
Sensitive personally identifiable information includes any data that could (i) be used to discriminate (ethnic heritage, religious preference, physical or mental health, for example), (ii) facilitate identity theft (like mother's maiden name), or (iii) permit access to a user's account (like passwords or PINs). Note that if the data described in this paragraph is not commingled with PII during storage or transfer, and it is not correlated with PII, then the data can be treated as Anonymous Data. If there is any doubt, however, the data should be treated as Sensitive PII. While not technically Sensitive PII, user data that makes users nervous (such as real-time location) should be handled in accordance with the rules for Sensitive PII.

Critical. *Release may create legal or regulatory liability for the organization.*

Important. *Release may create high risk of negative reaction by privacy advocates or damage the organization's image.*

Moderate. *Some user concerns may be raised, some privacy advocates may question, but repercussion will be limited.*

Low. *May cause some user queries. Scrutiny by privacy advocates unlikely.*

temporary storage
Temporary storage of data means that the data is only available while the application is running.

Appendix N: SDL Security Bug Bar (Sample)

Note: This sample document is for illustration purposes only. The content presented below outlines basic criteria to consider when creating security processes. It is not an exhaustive list of activities or criteria and should not be treated as such.

Please refer to the definitions of terms in this section.

Server Please refer to the Denial of Service Matrix for a complete matrix of server DoS scenarios. The server bar is usually not appropriate when user interaction is part of the exploitation process. If a Critical vulnerability exists only on server products, and is exploited in a way that requires user interaction and results in the compromise of the server, the severity may be reduced from Critical to Important in accordance with the NEAT/data definition of extensive user interaction presented at the start of the client severity pivot.	

Critical	Server summary: Network worms or *unavoidable* cases where the server is "owned."
	• Elevation of privilege: The ability to either execute arbitrary code or obtain more privilege than authorized.
	  • Remote anonymous user
	  • Examples:
	  • Unauthorized file system access: arbitrary writing to the file system
	  • Execution of arbitrary code
	  • SQL injection (that allows code execution)
	  • All write access violations (AV), exploitable read AVs, or integer overflows in remote anonymously callable code

Important	Server summary: Non-default critical scenarios or cases where mitigations exist that can help *prevent* critical scenarios.
	• Denial of service: Must be "easy to exploit" by sending a small amount of data or be otherwise quickly induced.
	  • Anonymous
	  • Persistent DoS
	  • Examples:
	  • Sending a single malicious TCP packet results in a Blue Screen of

Death (BSoD)

- Sending a small number of packets that causes a service failure

- Temporary DoS with amplification
  - Examples:
    - Sending a small number of packets that causes the system to be unusable for a period of time
    - A web server (like IIS) being down for a minute or longer
    - *A single remote client* consuming all available resources (sessions, memory) on a server by establishing sessions and keeping them open
- Authenticated
  - Persistent DoS ***against a high value asset***
    - Example:
      - Sending a small number of packets that causes a service failure for a ***high value asset*** in server roles (certificate server, Kerberos server, domain controller), such as when a domain-authenticated user can perform a DoS on a domain controller
- Elevation of privilege: The ability to either execute arbitrary code or to obtain more privilege than intended.
  - Remote authenticated user
  - Local authenticated user (Terminal Server)
    - Examples:
      - Unauthorized file system access: arbitrary writing to the file system
      - Execution of arbitrary code
  - All write AVs, exploitable read AVs, or integer overflows in code that can be accessed by remote or local authenticated users that are not administrators (Administrator scenarios do not have security concerns by definition, but are still reliability issues.)
- Information disclosure (targeted)
  - Cases where the attacker can locate and read information *from anywhere* on the system, including system information that was not intended or designed to be exposed

- Examples:
  - Personally identifiable information (PII) disclosure—see the Microsoft Privacy Standard for Development (MPSD) for detailed definitions and examples of PII
    - Disclosure of PII (email addresses, phone numbers, credit card information)
    - Attacker can collect PII without user consent or in a covert fashion

- Spoofing
  - An entity (computer, server, user, process) is able to masquerade ***as a specific entity*** (user or computer) of his/her choice.
    - Examples:
      - Web server uses client certificate authentication (SSL) improperly to allow an attacker to be identified as any user of his/her choice
      - New protocol is designed to provide remote client authentication, but flaw exists in the protocol that allows a malicious remote user to be seen as a different user of his or her choice
- Tampering
  - Modification of any "***high value asset***" data in ***a common or default scenario*** where the modification persists after restarting the affected software
  - Permanent or persistent modification of any user or system data used ***in a common or default scenario***
    - Examples:
      - Modification of application data files or databases in a common or default scenario, such as authenticated SQL injection
      - Proxy cache poisoning in a common or default scenario
      - Modification of OS or application settings without user consent in a common or default scenario
- Security features: Breaking or bypassing any security feature provided.

  Note that a vulnerability in a security feature is rated "Important" by default, but the rating may be adjusted based on other considerations as documented in the SDL bug bar.
  - Examples:
    - Disabling or bypassing a firewall without informing users or gaining consent

	• Reconfiguring a firewall and allowing connections to other processes
Moderate	• Denial of service • Anonymous • Temporary DoS without amplification in a default/common install • Example: • ***Multiple remote clients*** consuming all available resources (sessions, memory) on a server by establishing sessions and keeping them open • Authenticated • Persistent DoS • Example: • Logged in Exchange user can send a specific mail message and crash the Exchange Server, and the crash is **not** due to a write AV, exploitable read AV, or integer overflow • Temporary DoS with amplification in a default/common install • Example: • Ordinary SQL Server user executes a stored procedure installed by some product and consumes 100% of the CPU for a few minutes • Information disclosure (targeted) • Cases where the attacker can easily read information on the system ***from specific locations***, including system information, which was not intended/designed to be exposed. • Examples: • Targeted disclosure of anonymous data—see the Microsoft Privacy Standard for Development (MPSD) for detailed definitions of *anonymous data* • Targeted disclosure of the existence of a file • Targeted disclosure of a file version number • Spoofing • An entity (computer, server, user, process) is able to masquerade as a different, random entity that cannot be specifically selected. • Example: • Client properly authenticates to server, but server hands back a session from another random user who happens to be connected to the server

	at the same time
	• Tampering
	  • Permanent or persistent modification of any user or system data **in a specific scenario**
	  • Examples:
	  • Modification of application data files or databases ***in a specific scenario***
	  • Proxy cache poisoning ***in a specific scenario***
	  • Modification of OS/application settings without user consent ***in a specific scenario***
	  • Temporary modification of data **in a common or default scenario** that does not persist after restarting the OS/application/session
	• Security assurances:
	  • A security assurance is either a security feature or another product feature/function that customers expect to offer security protection. Communications have messaged (explicitly or implicitly) that customers can rely on the integrity of the feature, and that's what makes it a security assurance. Security bulletins will be released for a shortcoming in a security assurance that undermines the customer's reliance or trust.
	  • Examples:
	  • Processes running with normal "user" privileges cannot gain "admin" privileges unless admin password/credentials have been provided via intentionally authorized methods.
	  • Internet-based JavaScript running in Internet Explorer cannot control anything the host operating system unless the user has explicitly changed the default security settings.

Low	• Information disclosure (untargeted)
	  • Runtime information
	  • Example:
	  • Leak of random heap memory
	• Tampering
	  • Temporary modification of data *in a specific scenario* that does not persist after restarting the OS/application

Client
Extensive user action is defined as:
• "User interaction" can only happen in client-driven scenario.
• Normal, simple user actions, like previewing mail, viewing local folders, or file shares, are not extensive user interaction.
• "Extensive" includes users manually navigating to a particular website (for example, typing in a URL) or by clicking through a yes/no decision.
• "Not extensive" includes users clicking through e-mail links.
• NEAT qualifier (*applies to warnings only).* Demonstrably, the UX is:
  • Necessary (Does the user really need to be presented with the decision?)
  • Explained (Does the UX present all the information the user needs to make this decision?)
  • Actionable (Is there a set of steps users can take to make good decisions in both benign and malicious scenarios?)
  • Tested (Has the warning been reviewed by multiple people, to make sure people understand how to respond to the warning?)
  Clarification: Note that the effect of extensive user interaction is not one level reduction in severity, but is and has been a reduction in severity in certain circumstances where the phrase extensive user interaction appears in the bug bar. The intent is to help customers differentiate fast-spreading and wormable attacks from those, where because the user interacts, the attack is slowed down. This bug bar does not allow you to reduce the Elevation of Privilege below Important because of user interaction.	

Critical	Client summary:
	• Network Worms or *unavoidable* common browsing/use scenarios where the client is "owned" **without** warnings or prompts.
	• Elevation of privilege (remote): The ability to either execute arbitrary code or to obtain more privilege than intended.
	  • Examples:
	  • Unauthorized file system access: writing to the file system
	  • Execution of arbitrary code *without* extensive user action

	• All write AVs, exploitable read AVs, stack overflows, or integer overflows in remotely callable code (***without*** extensive user action)

Important	Client summary:
	• Common browsing/use scenarios where client is "owned" **with** warnings or prompts, or via extensive actions without prompts. Note that this does not discriminate over the quality/usability of a prompt and likelihood a user might click through the prompt, but just that a prompt of some form exists.
	• Elevation of privilege (remote)
	  • Execution of arbitrary code *with* extensive user action
	  • All write AVs, exploitable read AVs, or integer overflows in ***remote*** callable code (***with*** extensive user action)
	• Elevation of privilege (local)
	  • Local low privilege user can elevate themselves to another user, administrator, or local system.
	  • All write AVs, exploitable read AVs, or integer overflows in ***local*** callable code
	• Information disclosure (targeted)
	  • Cases where the attacker can locate and read information on the system, including system information that was not intended or designed to be exposed.
	  • Examples:
	  • Unauthorized file system access: reading from the file system
	  • Disclosure of PII—see the Microsoft Privacy Standard for Development (MPSD) for detailed definitions and examples of PII
	  • Disclosure of PII (email addresses, phone numbers)
	  • Phone home scenarios
	• Denial of service
	  • System corruption DoS requires re-installation of system and/or components.
	  • Example:
	  • Visiting a web page causes registry corruption that makes the machine unbootable
	  • Drive-by DoS

- Criteria:
  - Un-authenticated System DoS
  - Default exposure
  - No default security features or boundary mitigations (firewalls)
  - No user interaction
  - No audit and punish trail
  - Example:
    - Drive-by Bluetooth system DoS or SMS in a mobile phone

- Spoofing
  - Ability for attacker to present a UI that is different from but visually identical to the UI that users *must rely on to make valid trust decisions* in a *default/common scenario*. A trust decision is defined as any time the user takes an action believing some information is being presented by a particular entity—either the system or some specific local or remote source.
    - Examples:
      - Displaying a different URL in the browser's address bar from the URL of the site that the browser is actually displaying in a ***default/common scenario***
      - Displaying a window over the browser's address bar that looks identical to an address bar but displays bogus data in a ***default/common scenario***
      - Displaying a different file name in a "Do you want to run this program?" dialog box than that of the file that will actually be loaded in a ***default/common scenario***
      - Display a "fake" login prompt to gather user or account credentials
- Tampering
  - Permanent modification of any user data or data used to make trust decisions in a common or default scenario that persists after restarting the OS/application
    - Examples:
      - Web browser cache poisoning
      - Modification of significant OS/application settings without user consent
      - Modification of user data

	• Security features: Breaking or bypassing any security feature provided
	  • Examples:
	  • Disabling or bypassing a firewall with informing user or gaining consent
	  • Reconfiguring a firewall and allowing connection to other processes
	  • Using weak encryption or keeping the keys stored in plain text
	  • AccessCheck bypass
	  • Bitlocker bypass; for example not encrypting part of the drive
	  • Syskey bypass, a way to decode the syskey without the password

Moderate	• Denial of service
	  • Permanent DoS requires cold reboot or causes Blue Screen/Bug Check
	  • Example:
	  • Opening a Word document causes the machine to Blue Screen/Bug Check
	• Information disclosure (targeted)
	  • Cases where the attacker can read information on the system *from known locations*, including system information that was not intended or designed to be exposed
	  • Examples:
	  • Targeted existence of file
	  • Targeted file version number
	• Spoofing
	  • Ability for attacker to present a UI that is different from but visually identical to the UI that users *are accustomed to trust* in *a specific scenario*. "Accustomed to trust" is defined as anything a user is commonly familiar with based on normal interaction with the operating system or application but does not typically think of as a "trust decision."
	  • Examples:
	  • Web browser cache poisoning
	  • Modification of significant OS/application settings without user consent
	  • Modification of user data

Low	• Denial of service
	  • Temporary DoS requires restart of application
	  • Example:
	  • Opening a HTML document causes Internet Explorer to crash
	• Spoofing
	  • Ability for attacker to present a UI that is different from but visually identical to the UI *that is a single part of a bigger attack scenario*
	  • Examples:
	  • User has to go a "malicious" web site, click on a button in spoofed dialog box, and is then susceptible to a vulnerability based on a different browser bug
	• Tampering
	  • Temporary modification of any data that does not persist after restarting the OS/application.Information disclosure (untargeted)
	  • Example:
	  • Leak of random heap memory

Definition of Terms

authenticated
Any attack which has to include authenticating by the network. This implies that logging of some type must be able to occur so that the attacker can be identified.

anonymous
Any attack which does not need to authenticate to complete.

client
Either software that runs locally on a single computer or software that accesses shared resources provided by a server over a network.

default/common
Any features that are active out of the box or that reach more than 10 percent of users.

scenario
Any features that require special customization or use cases to enable, reaching less than 10 percent of users.

server
Computer that is configured to run software that awaits and fulfills requests from client processes that run on other computers.

Critical. *A security vulnerability that would be rated as having the highest potential for damage.*

Important. *A security vulnerability that would be rated as having significant potential for damage, but less than Critical.*

Moderate. *A security vulnerability that would be rated as having moderate potential for damage, but less than Important.*

Low. *A security vulnerability that would be rated as having low potential for damage.*

targeted information disclosure
Ability to intentionally select (target) desired information.

temporary DoS
A temporary DoS is a situation where the following criteria are met:

- The target cannot perform normal operations due to an attack.
- The response to an attack is roughly the same magnitude as the size of the attack.
- The target returns to the normal level of functionality shortly after the attack is finished. The exact definition of "shortly" should be evaluated for each product.

For example, a server is unresponsive while an attacker is constantly sending a stream of packets across a network, and the server returns to normal a few seconds after the packet stream stops.

temporary DoS with amplification
A temporary DoS with amplification is a situation where the following criteria are met:

- The target cannot perform normal operations due to an attack.
- The response to an attack is magnitudes beyond the size of the attack.
- The target returns to the normal level of functionality after the attack is finished, but it takes some time (perhaps a few minutes).

For example, if you can send a malicious 10-byte packet and cause a 2048k response on the network, you are DoSing the bandwidth by amplifying our attack effort.

permanent DoS
A permanent DoS is one that requires an administrator to start, restart, or reinstall all or parts of the system. Any vulnerability that automatically restarts the system is also a permanent DoS.

Denial of Service (Server) Matrix

Authenticated vs. Anonymous attack	Default/Common vs. Scenario	Temporary DoS vs. Permanent	Rating
Authenticated	Default/Common	Permanent	Moderate
Authenticated	Default/Common	Temporary DoS with amplification	Moderate
Authenticated	Default/Common	Temporary DoS	Low
Authenticated	Scenario	Permanent	Moderate
Authenticated	Scenario	Temporary DoS with amplification	Low
Authenticated	Scenario	Temporary DoS	Low
Anonymous	Default/Common	Permanent	Important
Anonymous	Default/Common	Temporary DoS with amplification	Important
Anonymous	Default/Common	Temporary DoS	Moderate
Anonymous	Scenario	Permanent	Important
Anonymous	Scenario	Temporary DoS with amplification	Important
Anonymous	Scenario	Temporary DoS	Low

Appendix O: Security Plan (Sample)

Note: This sample document is for illustration purposes only. The content presented below outlines basic criteria to consider when creating security processes. It is not a complete list of activities or criteria and should not be treated as such.

This document outlines the security activities for <SAMPLE> as they relate to each step of the Security Development Lifecycle (SDL). It describes the objective and provides a basic outline of each activity. It also identifies owners and expected deliverables from the activity. Most of the deliverables are included as exit criteria for different milestones for the project.

For successful execution of this security plan, the security team must perform security sign-offs, reviews, check-pointing, among other activities, for the security ship-readiness of the product at various project milestones. It is recommended that a "virtual" team be created, made up of individuals from program management, development, test, and UX.

The remainder of this document describes the minimum activities needed to build a secure product, the milestones during which they should be performed, and the owners and the deliverables for each activity.

Pre-SDL Requirements: Security Training

Education and Awareness

- Define which types of security training are available for your personnel.
- Identify who creates and/or delivers the classes.
- Define where they can find information about the classes.

Phase One: Requirements

Project Inception

- Determine whether the SDL applies to your component.
- Identify the team or individual that is responsible for tracking and managing security for your project.
- Ensure that bug reporting tools can track issues and that a database can be queried dynamically for all security issues at any time.
- Define and document a project's security bug bar.

Cost Analysis

- Complete a security risk assessment.

Phase Two: Design

Establish and Follow Best Practices for Design

- Create functional specifications that describe security features that are directly exposed to users.
- Design specifications should describe how to implement these features, and how to implement all functionality as secure features.

Risk Analysis

- Review security requirements and expectations to identify security concerns.
- Complete threat models for all functionality identified during the cost analysis phase.

Phase Three: Implementation

Creating Documentation and Tools

- Define security best practices documentation and tools.

Establish and Follow Best Practices for Development

- Establish, communicate, and follow effective practices for developing secure code to detect and remove security issues early in the development cycle.

Phase Four: Verification

Security and Privacy Testing

- Define fuzz testing, penetration, and run-time tests.
  - Determine which file formats will be fuzz tested.
  - Determine which networking interfaces will be fuzz tested.
  - Determine which tools will be used to fuzz test files and networking interfaces.
- Re-evaluate attack surface.
- Re-evaluate threat models.

Security Push

- Determine if there is a need for a security push.
- Define the security push steps.
  - Determine the timeline for the security push.
  - Determine whether there will be intensive education of project staff prior to the push.
  - Determine whether there are any other intensive tasks you want to focus on.
  - Define how the security vulnerabilities will be tracked.

Public Release Privacy Review

- Update the appropriate SDL Privacy Questionnaire for any significant privacy changes that were made during implementation verification.
- Work with your privacy advisor and legal representatives to create an approved privacy disclosure.
- Post the privacy disclosure to the appropriate website before each public release.

Phase Five: Release

Planning

- Identify who is responsible for security servicing.
- Provide contact information for people who respond to security incidents.
- Ensure process are in place to handle all types of security issues—for example, code reused or inherited from other teams and code licensed from third parties.
- Create a documented sustaining model that addresses the need to release immediate patches in response to security vulnerabilities and does not depend entirely on infrequent service packs.

Final Security Review and Privacy Review

- Define a due date for all project information that is required to start the Final Security Review (FSR).
- Review threat models.
- Review security issues that were deferred or rejected for the current release.
- Validate results of all security tools.
- Submit exception requests to a security advisor for review.

Release to Manufacturing/Release to Web

- Submit symbols for all publicly released products as part of the release process.
- Design and implement a sign-off process to ensure security and other policy compliance before you ship.

Post-SDL Requirement: Response

Security Servicing and Response Execution

- Develop a response plan that includes preparations for potential post-release issues.
- Be available to respond to any possible security vulnerabilities that warrant a response.

Appendix P: SDL-Agile Every-Sprint Requirements

Title	Requirement/ Recommendation	Applies to Online Services	Applies to Managed Code	Applies to Native Code
AllowPartiallyTrustedCallersAttribute (APTCA) review	Requirement		X	
Apply input validation (LOB)	Requirement	X	X	X
Annotate pointers to non-const parameters using Standard Annotation Language (SAL)	Requirement			X
Avoid Exec in stored procedures	Requirement	X		
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
Conduct internal security design review (LOB)	Requirement	X	X	X
Do not use banned APIs in new code	Requirement	X		X
Employ reflection and authentication relay defense	Requirement		X	X
Encrypt all secrets, such as credentials, keys, and passwords (LOB)	Requirement	X	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

Title	Requirement/ Recommendation	Applies to Online Services	Applies to Managed Code	Applies to Native Code
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
Harden or disable XML entity resolution	Requirement		X	X
Host security deployment review (LOB)	Requirement	X	X	X
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
Mitigate against cross-site request forgery (CSRF)	Requirement		X	
Mitigate against cross-site scripting (XSS)	Requirement	X	X	X
Secure sensitive data-at-rest (LOB)	Requirement	X	X	X
Secure sensitive data-in-transit (LOB)	Requirement	X	X	X
Update threat models for new features	Requirement	X	X	X
Use HeapSetInformation	Requirement			X

Title	Requirement/ Recommendation	Applies to Online Services	Applies to Managed Code	Applies to Native Code
Use safe integer arithmetic for memory allocation for new code	Requirement			X
Use safe redirect	Requirement	X	X	X
Use secure cookie over HTTPS	Requirement	X	X	X
Use standard annotation language (SAL) to annotate all functions	Recommendation	X		X
Use the most secure ATL version and secure COM coding requirements	Requirement			X
Use the /robust MIDL compiler switch	Requirement			X
Use the Relying Party Suite SDK	Requirement		X	X
Utilize LOB Secure Code Review (LOB)	Requirement	X	X	X
Avoid JavaScript eval function and equivalents	Recommendation	X		
Canonicalize URLs	Recommendation	X	X	X
Employ COM best practices	Recommendation			X
Encode long-lived pointers	Recommendation	X		X
Restrict database permissions	Recommendation	X		
Review error messages to ensure sensitive information is not disclosed	Recommendation	X	X	X
Use strict /GS option	Recommendation	X		X
Use transport layer encryption securely	Recommendation	X	X	X
Use whitelist of allowed domains to perform redirects	Recommendation	X	X	X

Appendix Q: SDL-Agile 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	
Ensure regular expressions must not execute in exponential time (O(2^n))	Requirement	X	X	X
Ensure sample code complies with appropriate SDL development practices	Requirement	X	X	X
Employ network fuzzing	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
Incorporate third-party component licensing security requirements in all new contracts	Requirement	X	X	X
Opt out of automatic MIME sniffing	Requirement		X	X
Use strongly named assemblies, and request minimal permissions	Requirement	X	X	
Apply no-open header to user-supplied downloadable files	Recommendation	X	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: Planning

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 R: SDL-Agile One-Time Requirements

Title	Requirement/ Recommendation	Applies to Online Services	Applies to Managed Code	Applies to Native Code
Avoid writable PE segments	Requirement	X		X
Create a baseline threat model	Requirement	X	X	X
Determine security response standards	Requirement	X	X	X
Do not use Visual Basic 6 to build products	Requirement	X	X	X
Establish a security response plan	Requirement	X	X	X
Identify primary security and privacy contacts	Requirement	X	X	X
Identify your team's privacy expert	Requirement	X	X	X
Identify your team's security expert	Requirement	X	X	X
Threat model your product, its attack surface, and its new features	Requirement	X	X	X
Use approved XML parsers	Requirement	X		X
Use latest compiler versions	Requirement	X	X	X
Use minimum code generation suite and libraries	Requirement		X	X
Configure bug tracking to track the cause and effect of security bugs	Recommendation	X	X	X
Remove dependencies on NTLM authentication	Recommendation	X	X	X

Appendix S: SDL-Agile 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 T: SDL-Agile 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.

Appendix U: SDL-LOB Risk Assessment Questionnaire

This sample document provides some criteria to consider when you build a risk assessment security questionnaire. It is not an exhaustive list and should not be treated as such. The weight assigned to an individual question depends on your business needs. Every question in each category can be mapped to a numeric score value, and all scores are then added together to identify the bucket that the application belongs to.

Introduction

The following questions are designed to help determine the risk rating of line-of-business (LOB) applications. The application team completes this questionnaire to assist in the determination of the risk rating. You can arrange these questions in categories, such as Architecture or Data Classification.

Risk Assessment Questionnaire

Audience

- What type of user access does your application offer (internal, external [Internet-facing], both, or neither)?

_____

- What is the basic authentication and authorization for the external-facing (Internet) portion of your application?

_____

- Are there anonymous users?

_____

- Is there a secure channel? What is that channel?

_____

Data Classification

- What type of data is contained in your application?

_____

- Does your application contain personal data?

_____

- How business-sensitive is the data managed by your application?

_____

Functionality

- What function does your application fulfill? How critical is its role?

_____

Architecture

- What is the authentication mechanism used by the client population?

_____

- Does your application have multiple user roles (for example, user and admin)?

_____

- Is code executed on the client machine (for example, ActiveX control, assembly)?

_____

- Where will your application be deployed?

_____

Process Control

- What type of source control do you use for your application?

_____

Privacy Release Issues

- Will the privacy statement or legal notice that was used in the existing application version change for this release? Is there a new privacy statement or legal notice available?

_____

Security Release Issues

- Does this version include changes to the authentication mechanism?

_____

- Does this web application or service provide functionality to other applications?

_____

Determine Your Security Impact Rating

The risk assessment is a quick way to determine your *security impact rating* and to estimate the work required to be compliant. The rating (High, Medium, Low) represents the degree of risk your LOB application presents from a security perspective. You need to complete only the steps that apply to your rating. For more detail, see the Line-of-Business section in the main Microsoft Security Development Lifecycle document.

Each company needs to define risk for their business and industry.

- Does this application handle personal information (employees, customers, business partners)?
- Does this application handle business sensitive data?
- Is this application key to providing a service or generating a product?
- Is this application key to running the business (finances, for example)?
- Who will have access to this application?

Appendix V: Lessons Learned and General Policies for Developing LOB Applications

The efforts of Microsoft IT to inventory, assess, and, if necessary, fix potential security issues that it discovers in its internal line-of–business (LOB) applications have proven to be successful. Microsoft IT has a much better grasp of the number and complexity of the applications that are used to run the day-to-day business at Microsoft. When Microsoft IT discovers any security issue in one application, it searches for that issue in other applications. Tightened security and privacy helps protect business-sensitive data and confidential employee data at Microsoft. Formalizing the security and privacy assessment process by means of SDL-LOB has raised the level of security and privacy awareness among the many internal development teams in Microsoft IT, reduced/identified risk, and improved the security of future development projects.

Lessons Learned

Procedural lessons learned as part of this include the following:

- Address security during application development. Waiting until the production phase to address security may expose vulnerabilities in the application.
- Create clearly written and easy-to-access documentation of security and privacy standards.
- Stabilize the process. Introducing constant changes to the standards or the process creates considerable churn and confusion.
- Use a process to prioritize which applications are examined or the order in which they are examined to help ensure that the most sensitive application/data is examined first.
- Develop a thoroughly considered process for tracking policy exceptions.
- Education is crucial to the success of a security and privacy program. Train developers, testers, and support personnel on an ongoing basis to provide up-to-date information.
- Security and privacy are ongoing concerns. Implement an experienced security team and a well-developed process to help ensure that applications incorporate ongoing changes.
- For third-party applications, a written statement from the vendor helps provide assurance that the software does not contain any hidden mechanisms that could be used to compromise or circumvent the software's security and privacy controls.
- Use an application portfolio management tool to track applications and to manage compliance with the overall governance process.
- The scope of security and privacy work may require a cross charge model that helps manage a balance between the availability of security/privacy subject matter experts (SMEs) against application release cycles.

Technical lessons learned as part of this include the following:

- Create security checklists that include step-by-step instructions for securing applications, hosts, and networks.
- Create privacy checklists that include systematic instructions for appropriate handling of applications that collect, use, or contain personal data (including notification requirements).
- Use a sound reporting solution to help drive compliance with the process.
- Ensure regular scanning of network and host to identify vulnerabilities, confirm patch management, and ensure regulatory compliance.
- Within a security tracking system, maintain an up-to-date inventory of the following items:
    - Applications, their versions, and the hosts (fully qualified domain name) they reside on
    - Compliance with SDL-LOB related tasks
    - Server lists (development, test, and production) by application
    - Policies and related standards
    - Exceptions to policies and standards

Setting Up a Security Team

Depending on the size of your organization, there may be a dedicated security team responsible for conducting reviews, setting standards, and monitoring compliance with regulatory requirements, standards, and policies. Specific responsibilities for this team may include:

- Security and privacy SMEs who conduct/monitor service delivery for the service levels.
- Account management that acts as a liaison with application teams, manage the application portfolio, and ensure that the process for SDL-LOB compliance runs smoothly.
- Remediation and risk management, which both prioritize applications for assessment and manage the remediation of high-risk vulnerabilities found during the assessment.
- Operations team that conducts network and host scanning post-assessment across the enterprise and production servers.
- Training and awareness for application teams to ensure that they comply with standards, policies, and best practices.
- Help desk to answer common questions and, as needed, escalate to the security and privacy SMEs.
- Authoring checklists, standards, and even corporate policy to meet security and privacy requirements.
- Resources, tools, and other content that assist application teams building low-risk or medium-risk LOB applications with security and privacy compliance.

In addition, there needs to be a security liaison within each development organization to ensure there is consistent messaging to the application teams and a single point of contact within the actual LOB organization.