Security Tsunami! SDL Fundamentals and Where to Start

Author(s)

SAFECode.org, Tania.Skinner@Intel.com

Abstract

In the world of secure development, there are 10,000 + things you can do… What are the fundamentals and where should you start? A healthy, secure development program is an ever-evolving suite of skills, tools and process coupled with an intricate understanding of an organization's capabilities, culture, and appetite for risk. SAFECode, an industry consortium of companies seeking, analyzing and sharing best practices in secure development, will present a set of broad secure development practices that have been effective in improving software security in real-world implementations across diverse product lines and various development methodologies. Learn about SAFECode member's key SDL practices and actionable guidance for how to prioritize and apply these fundamentals to your organization.

This paper is an abbreviated version of the SAFECode publication "Fundamental Practices for Secure Software Development" that is the result of best practice sharing amongst SAFECode members including Intel, Microsoft, Boeing, Siemens, Adobe, Dell EMC, Security Compass, Symantec, Veracode, Manicode and CA Technologies. The full paper has many contributors from across SAFECode member companies.

Biography

SAFECode.org: The Software Assurance Forum for Excellence in Code (SAFECode) is a non-profit organization exclusively dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods. SAFECode is a global, industry-led effort to identify and promote best practices for developing and delivering more secure and reliable software, hardware and services.

Tania Skinner graduated from the University of Illinois at Urbana-Champaign in 1992 with a BS in Computer Science. She joined Intel as a Software Engineer developing software for transistor level reliability testing and analysis. Through the years Tania has filled many roles on software projects from developer to database administrator to project manager. She eventually landed in the Corporate Quality group where she was responsible for leading a variety of cross-corporation SW development measurement and process improvement initiatives. Today, Tania is a Product Security Strategist & Researcher, responsible for leading and continuously improving Intel's Security Development Lifecycle (SDL) for all product development (including all HW, SW, FW development). Tania is also Intel's representative on SAFECode's Technical Leadership Council. Outside of Intel, Tania is Galactic Central Command to the very busy Skinner family. Tania specializes in secure development, software engineering, benchmarking, continuous improvement and the art of herding cats.

John Martin, CISSP, CISM, is Boeing's Security Program Manager with responsibilities ranging from DevSecOps to Commercial Software Security. His career spans the years between Blue-Box MF generators, through the era of automated hacks, and into our modern age of industrialized paranoia. Unlike more statesman-like peers, he suffers the disadvantage of being alive. He is, unfortunately, a frequent speaker on the topic of commercial software security. In his spare time, he designs specialized NSA-proof tin-foil hats designed to keep the implant signals in. John was named by SANS as one of the 10 Difference Makers in security for 2016. John represents Boeing on SAFECode's Technical Leadership Council.

1 Introduction

In developing the most recent version of the SAFECode Fundamental Practices for Secure Software Development paper[1], it became obvious to the team of contributors that the paper may seem intimidating to those who are new to the world of secure development. Those experienced in applying and evolving secure development practices will appreciate the depth of the full publication, but those who are just dipping their toes into the SDL ocean may find the details overwhelming and struggle with 'where do I start?' This shortened paper, presented in layman development terms, will attempt to

- demystify the secure development practices for the beginning secure development professional
- provide actionable guidance on where to start in applying secure software development practices
- provide guidance on planning the rollout of an SDL in your organization.

Secure development practices, or SDL (Security Development Lifecycle) as commonly referred to in the industry, fundamentally boil down to a few essential steps:

1. **Know what you need to protect, and protect it**
2. **Use secure code and components**
3. **Validate that protections are successful**
4. **Manage issues**
5. **Continuously improve**

Every SDL needs to address each of the steps listed above – with varying levels of breadth and depth. An SDL needs to cover all of the steps identified above, but doesn't initially need to address all of the specific techniques identified within each of these topic areas. Don't expect to start with a highly mature set of secure development practices, as there is a lot to do, but rather start with some solid foundational pieces and plan on constantly evolving and maturing your practices.

2 Know what you need to protect, and protect it

2.1 Know what you need to protect

A key consideration in applying secure development practices is understanding what, if anything, needs to be protected. Consider software that manages personal medical records. Clearly there is some very important information to protect and the software must be designed and implemented appropriately. Conversely, a simple text-editor could probably leverage the security of the underlying operating system. The first step in understanding what secure development practices to apply, is understanding what you need to protect. This is best achieved through a common secure development practice called Threat Modeling. Threat modeling enables the development team to identify data that needs to be protected, the ways in which an attacker could access the data, and the security features that are necessary to protect the data. These security features perform functions such as user identification and authentication, authorization of user access to information and resources, and auditing of users' activities as required by the specific application.

Threat modeling is started early in the development lifecycle before the core architecture or design is finalized and requires the participation of the system's architects and a security expert. During threat modeling, specific architectural or design changes will be identified to protect the data. There are many different models and methods for doing threat modeling.

Further Reading:

- More information about the benefits of threat modeling, some of the methodologies in use, simple examples and some of the pitfalls encountered in day-to-day practical threat modeling, as well as more complete references, may be found in the SAFECode paper "Tactical Threat Modeling." [2]

- Adam Shostack provides some very basic and layman friendly helpful information introducing threat modeling at https://misti.com/infosec-insider/threat-modeling-what-why-and-how.[4]

2.2 Protect It

Once you understand what information you have to protect and you have identified some methods for protecting it, you must apply secure design principles to the product's design. It is critical that the product architecture/design for any product is evaluated from the security perspective. Architecture/design flaws often cannot be fixed with a few lines of code, but rather require a re-architecture, a typically expensive proposition. Experienced practitioners have learned the hard way how best to design a product to prevent these kinds of bug. These practitioners have developed volumes of secure design principles and practices to guide your application's design. These include things like fail safely, the principle of least privilege, an encryption strategy, and defense in-depth, to name a few. Learn from others past mistakes and avoid them by applying secure design principles and practices. Make sure that all of your organization's architects and designers are well versed in the secure design principles and practices. Architecture/Design reviews should include a checklist of secure design principles and practices that designs are reviewed against. Some obvious items to include on your checklist are:

- Never trust input from an unknown component
- Always use secure protocols
- Never roll your own secure algorithm or crypto

Further Reading:

- More information on secure design principles and practices is provided in the full SAFECode Fundamental Practices for Secure Software Development[1], see section: Design.
- The principles of secure system design were first articulated in a 1974 paper by Jerome Saltzer and Michael Schroeder (The Protection of Information in Computer Systems)[5]
- ISO/IEC 27034-1 provides a method for identifying and incorporating Application Security Controls into a software project.[9]

3 Use secure code and components

Whether software is built as a monolithic block of code or as a highly modularized framework of components and micro-services, the following secure-coding principles are critical to achieving a well-written, secure software product.

3.1 Secure Coding

When developers write software they can make mistakes. Left undetected, these mistakes can lead to vulnerabilities that can compromise that software or the data it processes. A goal of developing secure software is to minimize the number of these unintentional code-level security vulnerabilities. This can be achieved by defining coding standards, selecting the most appropriate (and safe) languages, frameworks and libraries, ensuring their proper use (especially use of their security features), and performing both automated and manual code vulnerability analysis.

3.1.1 Establish Coding Standards and Conventions

Every software development language has strengths and weaknesses that need to be understood by the development team. For example, C++ requires significant memory management and input validation but there are also many commercial and Open Source code vulnerability analysis tools that can help with reviewing the code. Conversely, Golang avoids most memory issues but has few automated code analysis and test tools. Once a language choice is made, appropriate coding standards and conventions that support both writing secure code and the re-use of built-in security features and capabilities should be created, maintained and communicated to the development team.

Where possible, use built-in security features in the frameworks and tools selected and ensure that these are on by default. This will help all developers address known classes of issues, systemically rather than individually.

3.1.2 Use Safe Functions Only

Many programming languages have functions and APIs whose security implications were not appreciated when initially introduced but are now widely regarded as dangerous. Although C and C++ are known to have many unsafe functions, many other languages have at least some functions that are challenging to use safely. For example, dynamic languages such as JavaScript and PHP have several functions that generate new code at runtime (eval, seTtimeout, etc.) and are a frequent source of code execution vulnerabilities.

Developers should be provided guidance on what functions to avoid and their safe equivalents within the coding standards. Additionally, there are resources such as Microsoft's freely available banned.h header file that, when included in a project, will cause usage of unsafe functions to generate compiler errors. It is quite feasible to produce code bases free of unsafe function usage.

3.1.3 Use Current Compiler and Toolchain Versions and Secure Compiler Options

Using the latest versions of compilers, linkers, interpreters and runtime environments is an important security practice. Development tools and compilers are constantly evolving and incorporating automated checks for newly discovered vulnerabilities. For example, many C and C++ compilers automatically offer compile-time and run-time defenses against memory corruption bugs. Such defenses can make it harder for exploit code to execute predictably and correctly.

Enable secure compiler options and do not disable secure defaults for the sake of performance or backwards compatibility. These protections are defenses against common classes of vulnerabilities and represent a minimum standard.

3.1.4 Use Code Analysis Tools to Find Security Issues Early

Tooling should be deployed to assist in identifying and reviewing the usage of dangerous functions. Many static analysis tools plug directly into the integrated development environment and helps developers find security bugs early and effectively without leaving their native development environment. Many of these tools will identify common issues such as unbounded functions, potential buffer overflows, etc.

3.1.5 Handle Data Safely

Input validation is one of the single most important coding practices. You have probably heard about SQL injections, a common attack that can be achieved when code accepting user input doesn't validate that the input satisfies pre-defined size and type constraints. All user-originated input should be treated as untrusted and handled consistently throughout the entire system. Additionally, input from micro-services and 3rd-party components should also be treated as untrusted. For example, in web applications the same input data may be handled differently by the various components of the technology stack; the web server, the application platform, other software components and the operating system. An attacker could potentially inject malicious data at any input. If this data is not handled correctly it can be transformed into executing code or unintentionally provide access to resources.

There are many different techniques for safely handling input, including encoding and data binding, and many different problem areas. This secure coding area is addressed in great detail in the SAFECode Fundamental Practices for Secure Software Development paper.

3.1.6 Handle Errors

All applications run into errors at some point. While errors from typical use will be identified during functional testing, it is almost impossible to anticipate all the ways an attacker may interact with the application. Given this, a key feature of any application is to handle and react to unanticipated errors in a controlled and graceful way, and either recover or present an error message.

While anticipated errors may be handled and validated with specific exception handlers or error checks, it is necessary to use generic error handlers or exception handlers to cover unanticipated errors. If these generic handlers are hit, the application should cease performing the current action, as it should now be assumed to be in an unknown state. The integrity of further execution against that action can no longer be trusted.

Error handling should be integrated into the logging approach, and ideally different levels of detailed information should be provided to users as error messages and to administrators in log files.

When notifying the user of an error, the technical details of the problem should not be revealed. Details such as a stack trace or exception message provide little utility to most users, and thus degrade their user experience, but they provide insight to the attacker about the inner workings of the application. Error messages to users should be generic, and ideally from a usability perspective should direct users to perform an action helpful to them, such as “We’re sorry, an error occurred, please try resubmitting the form.” or “Please contact our support desk and inform them you encountered error 123456.” This gives an attacker very little information about what has gone wrong and directs legitimate users on what to do next.

3.2 Use Secure Components

Today’s modern applications contain a vast amount of third-party components. The majority of these third-party components are open source components.

Open source components are here to stay, and you must have a plan for 1) identifying components used, 2) assessing the security risk of these components, 3) monitoring components used, and 4) mitigating vulnerable components in your products. The management of security risk in third-party components is a complicated topic with many different approaches from highly governed internal repositories of acceptable components to security risk evaluations of components to automated scanning tools and scheduled security releases. SAFECode has an entire publication dedicated to this topic. Whatever the approach you take, you need to understand the risks and have a plan for selection, identification, monitoring and maintenance.

Further reading:

- SAFECode publication: Managing the risks inherent in the use of third-party components[2]

4 Validate that Protections are Successful

An essential component of an SDL program, and typically the first set of activities adopted by an organization, is some form of security testing. For organizations that do not have many security development practices, security testing is a useful tool to identify existing weaknesses in the product or service and serve as a compass to guide initial security investments and efforts, or to help inform a decision on whether or not to use third-party components. For organizations with mature security practices, security testing is a useful tool both to validate the effectiveness of those practices, and to catch flaws that were still introduced.

There are several forms of security testing and validation, and most mature security programs employ multiple forms. Broadly, testing can be broken down into automated and manual approaches, and then further categorized within each approach. There are tradeoffs with each form of testing, which is why most mature security programs employ multiple approaches.

As in other areas of development, automated security analysis allows for repeatable tests done rapidly and at scale. There are several commercial or free/open source tools that can be run either on developers' workstations as they work, as part of a build process on a build server or run against the developed product to spot certain types of vulnerabilities. Automated tools are adopted because of the speed and scale at which they run, and despite having false-positives, they allow organizations to focus their manual testing on their riskiest components, where they are needed most.

There are many categories of automated security testing tools. Some common automated testing tool categories: Static Analysis Security Testing (SAST), Dynamic Analysis Security Testing (DAST), fuzzing, software composition analysis, network vulnerability scanning and tooling that validates configurations and platform mitigations. Each category will be briefly introduced with an assessment of ease to implement for organizations newly venturing into this space. Note that no single tool will find all security flaws and mature organizations use a suite of tools.

The full publication SAFECode Fundamental Practices for Secure Software Development includes much greater detail and depth on security validation tools and methods. Below is a summary of some of the techniques and tools that may be utilized for security validation.

Table 1 Summary of Security Validation Techniques

Technique	**Type**	**Difficulty to Implement**	**Security Expertise Required**	**Cost**
Static Code Analysis	Automated	Easy	Low	$$
Dynamic Analysis Scanning	Automated + configuration	Easy	Low	$
Fuzz Parsers	Automated + configuration	Difficult	High	$$$
Perform Automated Functional Testing of Security Features/Mitigations	Automated	Varies	Low	$
Manual Verification of Security Features/Mitigations	Manual	Varies	Moderate	$$
Verify Secure Configurations and Use of Platform Mitigations	Automated	Easy	Some	$
Penetration Testing	Manual	Difficult	High	$$$

4.1 Use Static Analysis Security Testing Tools

PROS: good coverage efficiently, finds problematic patterns, can be tuned

CONS: may not cover all languages, can be noisy (false positives)

Static analysis is a method of inspecting either source code or the compiled intermediate language or binary component for flaws. It looks for known problematic patterns based simply on the application logic, rather than on the behavior of the application while it runs. SAST logic can be as simple as a regular expression finding a banned API via text search, or as complex as a graph of control or data flow logic that seems to allow for tainted data to attack the application. SAST is most frequently integrated into build automation to spot vulnerabilities each time the software is built or packaged; however, some offerings integrate into the developer environment to spot certain flaws as the developer is actively coding. SAST tends to give very good coverage efficiently, and with solutions ranging from free or open source security linters to comprehensive commercial offerings, there are several options available for an organization regardless of its budget.

4.2 Perform Dynamic Analysis Security Testing

PROS: ease of use, few false positives, good coverage

CONS: un-credentialed scans may produce false negatives

Dynamic analysis security testing runs against an executing version of a program or service, typically deploying a suite of prebuilt attacks in a limited but automated form of what a human attacker might try. These tests run against the fully compiled or packaged software as it runs, and therefore dynamic analysis is able to test scenarios that are only apparent when all of the components are integrated. Most modern websites integrate interactions with components that reside in separate source repositories – web service calls, JavaScript libraries, etc. – and static analysis of the individual repositories would not always be able to scrutinize those interactions. Additionally, as DAST scrutinizes the behavior of software as it runs, rather than the semantics of the language(s) the software is written in, DAST can validate software written in a language that may not yet have good SAST support.

4.3 Fuzz Parsers

PROS: deep test capabilities of known APIs and I/O

CONS: laborious to setup and determine root causes

Fuzzing, a specialized form of DAST (or in some limited instances, SAST) is the act of generating or mutating data and passing it to an application's data parsers (file parser, network protocol parser, inter-process communication parser, etc.) to see how they react. Fuzzing is a very advanced security validation technique.

4.4 Verify Secure Configurations and Use of Platform Mitigations

PROS: ease of use, few false positives, good coverage

CONS: un-credentialed scans may produce false negatives

Most security automation seeks to detect the presence of security vulnerabilities, but as outlined in the secure coding practices section, it is also important that software make full use of available platform mitigations, and that it configure those mitigations correctly. These tools are very strong in finding security issues like patch levels, secure configuration settings, secure and HTTP Only cookie flags, and that a server's SSL/TLS configuration is free of insecure protocol versions and cipher suites. Regular validation that software and services are correctly using available platform mitigations is usually far simpler to deploy and run than other forms of security automation, and the findings have very low false positive rates.

4.5 Perform Automated Functional Testing of Security Features/Mitigations

PROS: utilizes existing functional validation capabilities

CONS: Should not be perceived as comprehensive security validation

Organizations that use unit tests and other automated testing to verify the correct implementations of general features and functionality should extend those testing mechanisms to verify security features and mitigations designed into the software. For example, if the design calls for security-relevant events to be logged, unit tests should invoke those events and verify corresponding and correct log entries for them. Testing that security features work as designed is no different from verifying that any other feature works as designed and should be included in the same testing strategy used for other functionality.

4.6 Perform Manual Verification of Security Features/Mitigations

PROS: Flexibility, leverage prior findings, good training tool

CONS: more laborious, not comprehensive security validation

Manual testing is generally more laborious and resource intensive than automated testing, and unlike automated testing it requires the same investment every time that it is performed in order to produce similar coverage. In contrast to automated testing, because human testers can learn, adapt and make leaps of inference, manual testing can evolve based on previous findings and subtle indicators of an issue.

Just as security features should be included in existing unit tests and other automated functionality verifications, they should be included in any manual functional testing or code review efforts that are performed. Organizations employing manual quality assurance should include verification of security features and mitigations within the test plan, as these can be manually verified in the same way that any non-security feature is verified. Organizations that rely more heavily on automated functional testing should still perform peer review and occasional manual spot-checks of security features to ensure that the automated tests were implemented correctly because mistakes in some security features are less likely to be caught and reported by users than mistakes that result in erroneous outputs. For example, a user is likely to report if a text box is not working but is not likely to identify and report that security logging is not working.

4.7 Perform Penetration Testing

PROS: can discover complicated vulnerabilities, mimics attackers methodologies and tools

CONS: the most laborious and time-consuming form of security testing, not comprehensive coverage

**Many consulting companies exist that specialize in penetration testing services. For security-critical applications, contracting out penetration testing may be the best option.*

Penetration testing assesses software in a manner similar to the way in which hackers look for vulnerabilities. Penetration testing can find the widest variety of vulnerabilities and can analyze a software package or service in the broader context of the environment it runs in, actions it performs, components it interacts with, and ways that humans and other software interact with it. This makes it well suited to finding business logic vulnerabilities. Skilled penetration testers can find vulnerabilities not only with direct observation, but also based on small clues and inferences. Using their experience testing both their current projects, and from all previous engagements, they also learn and adapt and are thus inherently more capable than the current state of automated testing.

However, penetration testing is the most laborious and time-consuming form of security testing, requires specialized expertise, scales poorly and is challenging to quantify and measure. Large development organizations may staff their own penetration test team but many organizations engage other companies to perform penetration testing. Whether using staff or consultant testers, it is appropriate to prioritize penetration testing of especially security-critical functions or components. Also, given the expense and limited availability of highly skilled penetration testers, performing penetration testing tends to be most suitable after other, less expensive forms of security testing are completed. The time a penetration tester would spend documenting findings that other testing methodologies can uncover is time that the penetration tester is not using to find issues that only penetration testing is likely to uncover.

Further Reading:

- More information on secure design principles and practices is provided in the full SAFECode Fundamental Practices for Secure Software Development[1], see section: Testing and Validation.

5 Manage Issues

5.1 Issues Discovered During Development

Practices such as threat modeling, third-party component identification, SAST, DAST, penetration testing, etc. all result in artifacts that contain findings related to the product's security (or lack thereof). The findings from these artifacts must be tracked and action taken to remediate, mitigate or accept the respective risk. While we all desire to ship defect free products, there will inevitably come a time when you will be faced with a decision on whether or not to ship a product with a vulnerability. Before you find yourself in this unpleasant situation, you should have a predefined policy and model for assessing risk and making an informed, risk based decision. This model should include things such as (this is not an exhaustive list):

- Severity of the vulnerability
- Likelihood of exploit
- Potential for brand damage
- Remediation options

It is essential that you adopt a consistent formal process for assessing the risk with a decision maker with the appropriate level of authority to accept the risk and a system to track such decisions and any closing actions needed.

Further Reading:

- More information on managing security issues is provided in the full SAFECode Fundamental Practices for Secure Software Development[1], see section: Manage Security Findings.

5.2 Issues Discovered in Released Software

Despite the best efforts of software development teams, vulnerabilities may still exist in released software that may be exploited to compromise the software or the data it processes. Therefore, it is necessary to have a vulnerability response and disclosure process to help drive the resolution of externally discovered vulnerabilities and to keep all stakeholders informed of progress. This is particularly important when a vulnerability is being publicly disclosed and/or actively exploited. The goal of the process is to provide customers with timely information, guidance and, where possible, mitigations or updates to address threats resulting from such vulnerabilities.

Many companies establish a dedicated team, called PSIRT (Product Security Incident Response Team) whose primary function is to respond to and manage security vulnerabilities in released product. There are many industry papers regarding vulnerability response and disclosure process and/or PSIRT that provide a wealth of information and guidance.

Further Reading:

- More information on managing security issues is provided in the full SAFECode Fundamental Practices for Secure Software Development, see section: Vulnerability Response and Disclosure.

- FIRST PSIRT Services Framework: https://www.first.org/education/FIRST_PSIRT_Service_Framework_v1.0[7]
- ISO/IEC 29147 [10] – Vulnerability disclosure (available as a free download). Provides a guideline on receiving information about potential vulnerabilities from external individuals or organizations and distributing vulnerability resolution information to affected customers
- ISO/IEC 30111 [11] – Vulnerability handling processes (requires a fee to download). Gives guidance on how to internally process and resolve reports of potential vulnerabilities within an organization

6 Continuously Improve

Secure development practices are constantly evolving as technology, tools and attack techniques evolve. It is essential that you establish feedback loops that evaluate the outputs of your secure development practices and identify improvement needed to your policies, processes and or tools. The Cost of Quality [8] model principles apply to security just as they do to quality-- security vulnerabilities found early in development cost a lot less to fix than those found later in development. We want to prevent security vulnerabilities from being introduced, but when they are present, we want to identify and fix them as early as possible. Performing root cause analysis on issues found will help us identify the gaps in process/policy and prevent them from re-occurring. An example:

- A buffer overflow vulnerability is identified in a released product with a root cause of improper usage of an unbounded function. POSSIBLE improvements: make sure static code analysis software scans for these by default and identifies them, update policies requiring teams to run static code analysis software and address findings, update coding standards to ban usage of unsafe, unbounded functions.

This type of analysis and secure development practice evolution is critical to identifying gaps and closing those gaps. In addition, as the secure development practices you have adopted become standard practice and are woven into development, you will want to assess other practices, policies or tools that can strengthen the security posture of your products and/or improve efficiency. If done well, your secure development practices should be constantly evolving, just as the attackers and attack techniques are.

7 Planning the Implementation and Deployment of Secure Development Practices

An organization's collection of secure development practices is commonly referred to as a Secure Development Lifecycle or SDL. While the set of secure development practices described previously in this paper are essential components of an SDL, they are not the only elements of a mature SDL. A healthy SDL includes all the aspects of a healthy business process, including program management, stakeholder management, deployment planning, metrics and indicators, and a plan for continuous improvement. Whether defining a new set of secure development practices or evolving existing secure development practices, there is a variety of factors to consider that may aid or impede the definition, adoption and success of the secure development program overall. Below are some items to consider in planning the implementation and adoption of an SDL:

- Culture of the organization
- Expertise and skill level of the organization
- Product development model and lifecycle
- Scope of the initial deployment
- Stakeholder management and communication
- Efficiency measurement

- SDL process health
- Value proposition for the secure development practices

7.1 Culture of the Organization

The culture of the organization must be considered when planning the deployment of any new process or set of application security controls. Some organizations respond well to corporate mandates from the CEO or upper management, while others respond better to a groundswell from the engineering team. Think about the culture of the organization that must adopt these practices. Look to the past for examples of process or requirements changes that were successful and those that were not. If mandates work well, identify the key managers who need to support and communicate a software security initiative. If a groundswell is more effective, think about highly influential engineering teams or leaders to engage in a pilot and be the first adopters.

7.2 Expertise and Skill Level of the Organization

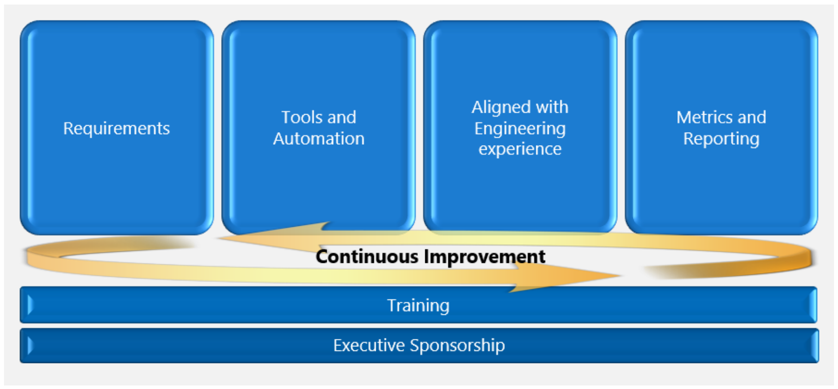
If an organization is to be successful in implementing an SDL, some level of training is necessary. The entire organization should be made aware of the importance of security, and more detailed technical training must be provided to development teams that clearly articulates the specific expectations of individuals and teams. If individuals do not understand why these practices are important and the impact of not performing these practices, they are less likely to support them. Additional training will likely be needed, depending on the expertise in the organization. For each of the secure development practices, consider the expertise/skill level needed and the coverage of that expertise in the organization.

7.3 Product Development Model and Lifecycle

Along with a specification of the secure development practices, it is essential to consider when the practices are required. When or how often a practice is applied is highly dependent on the development model in use and the automation available. Although security practices executed in sequential and logical order can result in greater security gains (e.g., threat model before code is committed) and cost effectiveness than ad hoc implementation, in agile or continuous development environments, other triggers such as time (e.g., conduct DAST monthly) or response to operating environment changes (e.g., deployment of a new service or dataset) should be considered.

Not everyone is nor needs to be a security engineer, and the SDL framework should provide engineers with clear actionable guidance, supported by efficient processes, carefully selected tools and intelligent automation, tightly aligned with the engineering experience. To facilitate change at scale, security process and tools must be integrated into the engineers' world and not exist as a separate security world, which is often seen as a compliance function. This not only helps to reduce friction with engineers, it also enables them to move faster by integrating security tools into the world in which they live and operate.

Table 2 SDL framework for all development models



7.4 Scope of Initial Deployment

Often, the teams implementing the secure development program are resource constrained and may need to consider different ways to prioritize the rollout across the organization. There are many ways to manage the rollout of the SDL. Having a good understanding of how the project planning process works and the culture of the organization will help the secure development program manager make wise choices for the implementation and adoption of secure development practices. Below are some options to consider:

- Will the initial rollout include all secure development practices or a subset?
- Consider the product release roadmap and select an adoption period for each team that allows them time to plan for the adoption of the new process. It may be unwise to try to introduce a new process and set of practices to a team nearing completion of a specific release.
- The initial rollout might also choose to target teams with products of higher security risk posture and defer lower risk products for a later start.
- Consider allowing time for transition to full adherence to all SDL requirements.

7.5 Stakeholder Management and Communications

Deploying a new process or set of practices often requires the commitment and support of many stakeholders. Identify the stakeholders, champions and change agents who will be needed to assist with the communications, influencing and roll-out of the program. Visit these stakeholders with a roadshow that clearly explains the value and commitment to secure development practices and what specifically they are being asked to do.

7.6 Compliance Measurement

In implementing a new set of security requirements, the organization must consider what is mandatory and what (if anything) is optional. Most organizations have a governance or risk/controls function that will require indicators of compliance with required practices. In defining the overall SDL program, consider the following:

- Are teams required to complete all of the secure development practices? What is truly mandatory? Is there a target for compliance?
- What evidence of practice execution is required?
- How will compliance be measured? What types of reports are needed?
- What happens if the compliance target is not achieved? What is the risk management process that should be followed? What level of management can sign off on a decision to ship with open exceptions to security requirements? What action plans will be required to mitigate such risks and how will their implementation be tracked?

7.7 SDL Process Health

A good set of secure development practices is constantly evolving, just as the threats and technologies involved are constantly evolving. The SDL process must identify key feedback mechanisms for identifying gaps and improvements needed. The vulnerability management process is a good source of feedback about the effectiveness of the secure development practices. If a vulnerability is discovered post-release, root-cause analysis should be performed. The root cause may be:

- A gap in the secure development practices that needs to be filled
- A gap in training/skills
- The need for a new tool or an update to an existing tool

In addition, development teams will have opinions regarding what is working and what is not. The opinions of development teams should be sought, as they may help identify inefficiencies or gaps that should be addressed. Industry reports regarding trends and shifts in secure development practices should be monitored and considered. A mature SDL has a plan and metrics for monitoring the state of secure development practices across the industry and the health of the organization's secure development practices.

7.8 Value Proposition

There will be times when the funding for secure development practices support will be questioned by engineering teams and/or management. A mature secure development program will have good metrics or indicators to articulate the value of the secure development practices to drive the right decisions and behaviors. Software security is a very difficult area to measure and portray. Some common metrics can include industry cost-of-quality models, where the cost of fixing a vulnerability discovered post-release with customer exposure/damage is compared to that of finding/fixing a vulnerability early in the development lifecycle via a secure development practice. Other metrics include severity and exploitability of externally discovered vulnerabilities (CVSS score trends) and even the cost of product exploits on the "grey market." There is no perfect answer here, but it is a critical aspect of a secure development program to ensure that the value proposition is characterized, understood and can be articulated in a way that the business understands.

Further Reading:

- More information on managing security issues is provided in the full SAFECode Fundamental Practices for Secure Software Development [1], see section: Planning the Implementation and Deployment of Secure Development Practices

8 Moving Industry Forward

One of the more striking aspects of SAFECode's work in creating this paper was an opportunity to review the evolution of software security practices and resources in the seven years since the second edition was published. Though much of the advancement is a result of innovation happening internally within individual software companies, SAFECode believes that an increase in industry collaboration has amplified these efforts and contributed positively to advancing the state of the art across the industry.

References

1. SAFECode. “Fundamental Practices for Secure Software Development, Third Edition”. SAFECode.org. https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf (accessed August 19, 2018)
2. SAFECode. “Tactical Threat Modeling”. SAFECode.org. https://www.safecode.org/wp-content/uploads/2017/05/SAFECode_TM_Whitepaper.pdf (accessed August 19, 2018)
3. SAFECode. “Managing Security Risks Inherent in the Use of Third-party Components”. SAFECode.org. https://www.safecode.org/wp-content/uploads/2017/05/SAFECode_TPC_Whitepaper.pdf (accessed August 19, 2018)
4. Shostack, Adam. “Threat Modeling: What, Why, and How?”. https://misti.com/infosec-insider/threat-modeling-what-why-and-how (accessed August 10, 2018)
5. Saltzer, Jerome and Schroeder, Michael. “The Protection of Information in Computer Systems”. MIT.edu. http://web.mit.edu/Saltzer/www/publications/protection/ (accessed October 2017)
6. ISO/IEC 27034-1 provides a method for identifying and incorporating Application Security Controls into a software project.
7. FIRST. “FIRST PSIRT Services Framework.” FIRST.org. https://www.first.org/education/FIRST_PSIRT_Service_Framework_v1.0 (accessed July 26, 2018)
8. ASQ. “Cost of Quality.” ASQ.org. http://asq.org/learn-about-quality/cost-of-quality/overview/overview.html (accessed August 10, 2018)
9. International Organization for Standardization. “ISO/IEC 27034-1”. ISO.org. https://www.iso.org/contents/data/standard/04/43/44378.html.
10. International Organization for Standardization. “ISO/IEC 29147”. ISO.org. https://www.iso.org/contents/data/standard/04/51/45170.html.
11. International Organization for Standardization. “ISO/IEC 30111”. ISO.org. https://www.iso.org/contents/data/standard/05/32/53231.html