SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

Fundamental Practices for Secure Software Development

Essential Elements of a Secure Development Lifecycle Program

Third Edition

March 2018

Table of Contents

Executive Summary

Software assurance encompasses the development and implementation of methods and processes for ensuring that software functions as intended and is free of design defects and implementation flaws. In 2008, the Software Assurance Forum for Excellence in Code (SAFECode) published the first edition of "SAFECode Fundamental Practices for Secure Software Development" in an effort to help others in the industry initiate or improve their own software assurance programs and encourage the industry-wide adoption of fundamental secure development practices. In 2011, a second edition was published, which updated and expanded the secure design, development and testing practices.

As the threat landscape and attack methods have continued to evolve, so too have the processes, techniques and tools to develop secure software. Much has been learned, not only through increased community collaboration but also through the ongoing internal efforts of SAFECode's member companies.

This, the third edition of "SAFECode Fundamental Practices for Secure Software Development," includes updates to the fundamental practices to reflect current best practice, new technical considerations and broader practices now considered foundational to a successful Secure Development Lifecycle (SDL) program.

- Requirement Identification
- Management of Third-party Component Components (both Open Source and Commercial Off-the-shelf)
- Security Issue Management
- Vulnerability Response and Disclosure

This paper also includes considerations for those planning and implementing a set of secure development practices, or, as commonly known, a Secure Development Lifecycle (SDL).

Although this version addresses more elements of a Secure Development Lifecycle, just as with the original paper, this paper is not meant to be a comprehensive nor exhaustive guide. Rather, it is meant to provide a foundational set of secure development practices that have been effective in improving software security in real-world implementations by SAFECode members across their diverse development environments and product lines.

It is important to note that these were identified through an ongoing collaboration among SAFECode members and are "practiced practices." By bringing these methods together and sharing them with the larger community, SAFECode hopes to help the industry move from "theoretical" best practices to those that are proven to be both effective and implementable.

Introduction

Following the publication of the SAFECode "Fundamental Practices for Secure Software Development, v2" (2011), SAFECode also published a series of complementary guides, such as "Practices for Secure Development of Cloud Applications" (with Cloud Security Alliance) and "Guidance for Agile Practitioners." These more focused guides aligned with the move toward more dynamic development processes and addressed some of the security concerns and approaches for web applications and cloud services. The pace of innovation continues to increase, and many software companies have transitioned away from multi-year development cycles in favor of highly iterative and more frequent releases, including some that release "continuously." Additionally, reliance on third-party components, both commercial and OSS, is growing, and these are often treated as black boxes and are reviewed with a different level of scrutiny from in-house developed software – a difference that can introduce risk. Add to this a need to be compliant with many standards and regulations, and software development teams can struggle to complete the necessary security activities.

Acknowledging these concerns, a review of the secure software development processes used by SAFECode members reveals that there are corresponding security practices for each activity in the software development lifecycle that can help to improve software security. These practices are agnostic about any specific development methodology, process or tool, and, broadly speaking, the concepts apply to the modern software engineering world as much as to the classic software engineering world.

The practices defined in this document are as diverse as the SAFECode membership, spanning cloud-based and online services, shrink-wrapped software and database applications, as well as operating systems, mobile devices, embedded systems and devices connected to the internet. The practices identified in this document are currently practiced among SAFECode members -- a testament to their ability to be integrated and adapted into a wide variety of real-world development environments -- and while each practice adds value, SAFECode members agree that to be effective, software security must be addressed throughout the software development lifecycle, rather than as a one-time event or single box on a checklist.

Audience

The guide is intended to help others in the industry initiate or improve their own software security programs and encourage the industry-wide adoption of fundamental secure development methods. Much of this document is built from the experience of large companies that build software that is used by many millions and in some cases billions of users. Small software companies should also be able to benefit from many of these recommendations.

Disclaimer: the practices presented herein focus on software development. Although these practices support meeting some legal or regulatory requirements, the practices themselves do not specifically address legal issues or some other aspects of a comprehensive security assurance approach, such as physical access to facilities or physical defenses of devices.

SAFECode Guidance and Software Assurance Programs

Software assurance cannot be achieved by a single practice, tool, heroic effort or checklist; rather it is the result of a comprehensive secure software engineering process that spans all parts of development from early planning through end of life. It is also important to realize that, even within a single organization and associated Secure Development Lifecycle (SDL), there is no one-size-fits-all approach. The SDL must be firm in its approach to security but flexible enough in its application to accommodate variations in a number of factors, including different technologies and development methodologies in use and the risk profile of the applications in question.

Every member of the organization plays a role in any effort to improve software security and all are rightfully subject to high expectations from customers. While each one of the practices described in subsequent sections can help an organization minimize the risk of vulnerabilities, a more holistic view is required. A key principle for creating secure code is the need for an organizational commitment starting with executive-level support, clear business and functional requirements, and a comprehensive secure software development lifecycle that is applicable throughout the product's lifecycle and incorporates training of development personnel. We believe that every technology developer has a responsibility to implement and take part in such a process. This is fundamental to achieving a "security culture" in a software organization. This paper describes fundamental practices for all roles that participate in software development.

Application Security Control Definition

Identifying and managing Application Security Controls (ASCs) or security requirements and security issues are essential aspects of an effective secure software development program. Clear and actionable technical controls that are continuously refined to reflect development processes and changes in the threat environment are the foundation upon which SDL tools and process are built. The practices identified in this document and application security controls they drive will lead to the identification of software design or implementation weaknesses, which when exploited expose the application, environment or company to a level of risk. These issues must be tracked (see Manage Security Findings) and action must be taken to improve the overall security posture of the product. Further, effective tracking supports the ability to both gauge compliance with internal policies and external regulations and define other security assurance metrics.

Actively Manage Application Security Controls

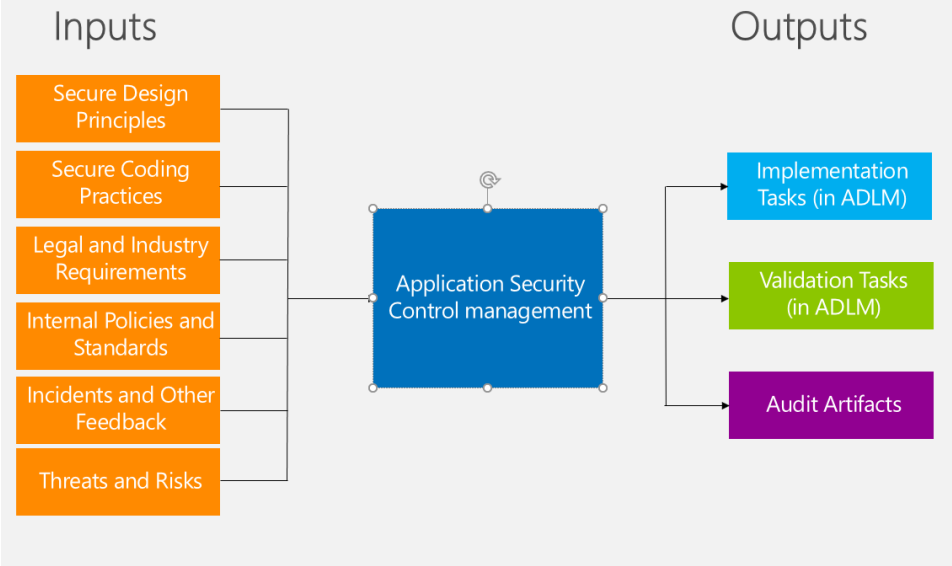
Regardless of the development methodology being used, defining application security controls begins in (or even before) the Design stage and continues throughout an application's lifecycle in response to changing business requirements and an ever-evolving threat environment.

The inputs used to identify the necessary security requirements[1] should include the secure design principles described in the following section and feedback from the established vulnerability management program, and may also require input from other stakeholders, such as a compliance team (e.g., if the application must comply with standards such as HIPAA, PCI, GDPR, etc.) or an operations and deployment team, because where and how the application is deployed may affect its security needs.

At a high level, the workflow should include:

1. Identifying threats, risks and compliance drivers faced by this application
2. Identifying appropriate security requirements to address those threats and risks
3. Communicating the security requirements to the appropriate implementation teams
4. Validating that each security requirement has been implemented
5. Auditing, if required, to demonstrate compliance with any applicable policies or regulations

[1] Security requirements and application security controls are used interchangeably throughout this document.



Application Security Control Management

Each security requirement identified should be tracked through implementation and verification. A best practice is to manage the controls as structured data in an Application Development Lifecycle Management (ADLM) system rather than in an unstructured document.

Design

The creation of secure software involves activities at a number of levels. If the organizations that will use the software have internal security policies or must comply with external laws or regulations, the software must incorporate security features that meet those requirements. In addition to incorporating security features, the architecture and design of the software must enable it to resist known threats based on intended operational environment.

The process of threat modeling, when combined with appropriate consideration of security requirements and the secure design principles, will enable the development team to identify security features that are necessary to protect data and enable the system to meet its users' requirements. These 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.

The fundamental practices described in this document primarily deal with assurance – with the ability of software to withstand attacks that attempt to exploit design or implementation errors such as buffer overruns (in native code) or cross-site scripting (in website code). In some cases, such as encryption and sensitive data protection, the selection or implementation of security features has proven to be sufficiently subtle or error-prone so that design or implementation choices are likely to result in vulnerabilities. The authors of this document have included recommendations for the security of those features with the goal of enabling organizations to produce systems that are secure from attack from any cause.

Secure Design Principles

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) The principles from that paper that have proven most important to the designers of modern systems are:

- **Economy of mechanism:** keep the design of the system as simple and small as possible.
- **Fail-safe defaults:** base access decisions on permission (a user is explicitly allowed access to a resource) rather than exclusion (a user is explicitly denied access to a resource).
- **Complete mediation:** every access to every object must be checked for authorization.
- **Least privilege:** every program and every user of the system should operate using the least set of privileges necessary to complete the job.
- **Least common mechanism:** minimize the amount of mechanism common to more than one user and depended on by all users.
- **Psychological acceptability:** it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.
- **Compromise recording:** it is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss.

The Saltzer and Schroeder principles set a high bar for the designers of secure systems: following them strictly is not a simple task. Nonetheless, designers who refer to them and attempt to follow their guidance are more likely to create systems that meet the goals of protecting information and resisting attack.

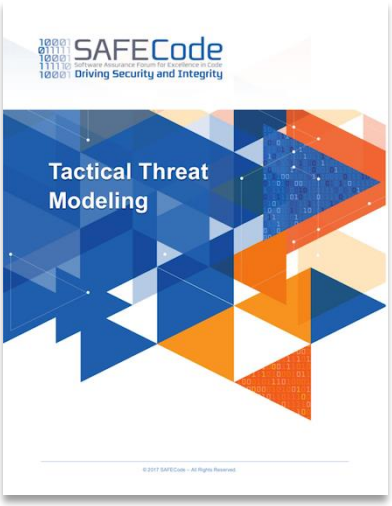
In the years since Saltzer and Schroeder published their paper, experience has demonstrated that some additional principles are important to the security of software systems. Of these, the most important are:

- **Defense in depth:** design the system so that it can resist attack even if a single security vulnerability is discovered or a single security feature is bypassed. Defense in depth may involve including multiple levels of security mechanisms or designing a system so that it crashes rather than allowing an attacker to gain complete control.
- **Fail securely:** a counterpoint to defense in depth is that a system should be designed to remain secure even if it encounters an error or crashes.
- **Design for updating:** no system is likely to remain free from security vulnerabilities forever, so developers should plan for the safe and reliable installation of security updates.

The principles described above are relevant to the design of any system, whether for client or server, cloud service, or Internet-of-Things device. The specifics of their application will vary – a cloud service may require multiple administrative roles, each with its own least privilege, while an IoT device will require special considerations of the need for security updates and of the need to fail securely and safely. But the principles are general and provide valuable security guidance for the designers and architects of all classes of systems.

Threat Modeling

Threat modeling is a security focused design activity and a fundamental practice in the process of building trusted technology; it has proven to be one of the best “return on investment” activities for identifying and addressing design flaws before their implementation into code.



The greatest benefit from threat modeling is realized when it is performed early in the development lifecycle before code is committed. Even if that cannot be achieved, threat modeling provides value in mapping out a system to understand and prioritize its weaknesses. Downstream activities such as static analysis, security testing and customer documentation can be greatly guided and focused based on the context provided by threat modeling.

There are many possible ways of generating a threat model, and the consensus is that there is no one single perfect way. A valid process is one that is repeatable and manageable, and above all one that can identify potential threats.

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."

Perform Architectural and Design Reviews

Architectural and design review should be incorporated into a security program. A poorly designed system that allows a malicious actor to fully compromise a system and its data through a design or logic

flaw can be catastrophic and difficult to remediate. The design review should be conducted with reference to the design principles above. To the extent that an architecture falls short of meeting those principles, it is likely to fall short of its goal of protecting information for its users.

Develop an Encryption Strategy

Encryption is the most common mechanism to protect data from unintended disclosure or alteration, whether the data is being stored or transmitted. While it is possible to retroactively build encryption into a feature, it is easier, more efficient and more cost-effective to consider encryption during the design process. Threat modeling, described earlier in this section, is a useful tool to identify scenarios that benefit from encryption; however, developing an encryption strategy (how to encrypt, store and manage encryption keys, etc.) is typically enough effort to be tackled as its own task. Most larger organizations benefit from a centralized encryption strategy designed, deployed and governed (via a review board) by experts in cryptography, rather than having individual teams pursue redundant and potentially incompatible or flawed efforts.

There are several key components of an encryption strategy:

- **Definitions of what to protect:** at the very least, all internet traffic should be encrypted while in transit, and it is recommended that, barring a compelling reason to the contrary, traffic within private networks should also be encrypted. When storing data, whether in a file, within cloud storage, within a database, or other persistent location, organizations should develop clear criteria for what types of data should be encrypted, and what mechanisms are acceptable to protect that data.

- **Designation of mechanisms to use for encryption:** despite the common usage of encryption, it remains quite challenging to implement correctly. There are numerous encryption algorithms, key lengths, cipher modes, key and initial vector generation techniques and usages, and cryptographic libraries implementing some or all of these functions. Making an incorrect choice for any one of these aspects of encryption can undermine the protection. Rather than having developers figure out the correct choice at the time of implementation, organizations should develop clear encryption standards that provide specifics on every element of the encryption implementation. Only industry-vetted encryption libraries should be used, rather than custom internal implementations, and only strong, unbroken algorithms, key lengths and cipher modes (for the specific scenarios) should be allowed.

  For encryption in transit, only strong versions of the encryption protocol should be allowed. For example, all versions of SSL are considered broken, and early versions of TLS are no longer recommended due to concerns about weaknesses in the protocol. There are several freely available tools[2] that can be used to verify that only strong versions of a transit encryption protocol are in use.

  For encryption at rest, in addition to the considerations above, there is a decision of how to deploy that encryption. Solutions such as disk encryption, OS credential/key managers, and database transparent encryption are relatively easy to deploy and provide protection against

[2] SSLlabs maintains a list of SSL/TLS assessment tools

offline attack. If the primary risk is theft of a device or disk, these solutions are typically the best option, as they require little to no custom application logic to implement, and most cloud and mobile platforms make enabling this form of storage encryption quite easy (or have it enabled by default). However, these solutions do not protect against any compromise of application logic and are insufficient protection for very sensitive or highly confidential data. For these scenarios the application must implement encryption specifically against the data prior to writing it to storage. As previously mentioned, threat modeling can be a useful tool to identify scenarios where protection solely against offline attack is insufficient, and where encryption implemented within application logic is justified.

- **Decide on a key and certificate management solution:** encrypting data is only one half of an encryption strategy. The other half is the solution to manage encryption keys and certificates. Every party that can access an encryption key or certificate is a party that can access the encrypted data, and so a solution to manage the keys and certificates should control who (whether a person, or a service) has access, and provide a clear audit log of that access. Additionally, keys and certificates have a limited lifespan, with the exact duration being decided during the development of the encryption strategy. The key and certificate management solutions should have a mechanism to manage the lifecycle of keys and certificates, providing mechanisms to deploy new keys and certificates once the previous ones are near expiration.

  Hard-coding encryption keys (or other secrets) within source code leaves them very vulnerable and must be avoided.

- **Implement with cryptographic agility in mind:** encryption algorithms might potentially be broken at any time, even if they are considered current best practices, and encryption libraries may have vulnerabilities that undermine otherwise sound algorithms. An encryption strategy should specify how applications and services should implement their encryption to enable transition to new cryptographic mechanisms, libraries and keys when the need arises.

Standardize Identity and Access Management

Most products and services have the need to verify which principal, either human user or other service or logical component, is attempting to perform actions and whether the principal in question is authorized to perform the action it is attempting to invoke. These actions should all be auditable and logged in the logging system being used. Authenticating the identity of a principal and verifying its authorization to perform an action are foundational controls that other security controls are built upon, and organizations should standardize on an approach to both authentication and authorization. This provides consistency between components as well as clear guidance on how to verify the presence of the controls. Threat modeling will help identify where authorization checks for access should be applied, and the secure design principles, such as least privilege, economy of mechanism, and complete mediation are fundamental to the design of robust Identity and access management mechanisms.

Several components comprise identity and access management, and the standard that an organization applies to its products and services should include:

- **The mechanism by which users (both end-users and organization administrators) authenticate their identities.** Many aspects must be considered in order to provide assurance that the users are who they say they are: how they initially sign up or are provisioned, the credentials (including multi-factor credentials) they provide each time they authenticate, how they restore their access to the system should they lose a necessary credential, how they authenticate when communicating to a help desk for the product, etc. An oversight in any of these elements can undermine the security of the authentication system, and by extension the whole product. If every feature team builds its own authentication system, there is a much higher likelihood that mistakes will be made.

  Organizations should delegate authentication to a third-party identity provider via a mechanism such as OpenID, incorporate into their product a prebuilt identity solution from an organization specializing in creating authentication technologies, or centralize the design and maintenance of an authentication solution with an internal team that has expertise specific to authentication.

- **The mechanism(s) by which one service or logical component authenticates to another, how the credentials are stored, and how they are rotated in a timely fashion.** Much like encryption keys, service credentials should not be stored within a source repository (neither hard-coded nor as a config file), as there are insufficient protections preventing the disclosure of those credentials. Credential rotation for services can be challenging if not designed into a system, as both the consumer and producer services need to synchronize the change in credentials.

- **The mechanism(s) that authorizes the actions of each principal.** One of the most explicit security tasks when building an application or service is developing a mechanism to control which actions each principal is allowed to perform. Many of the secure design principles previously discussed apply directly to authorization. Specifically, an authorization strategy should be based on complete mediation, where every access or action against every object is checked for authorization. The safest way to ensure this is to build the application such that, without an explicit authorization check, the default state is to deny the ability to perform the access or action (an approach commonly referred to as “default deny”). This approach makes it far easier to detect when there is an oversight in applying authorization, and the result of the oversight is to put the system in an overly restrictive rather than overly permissive state. Additionally, an authorization strategy should be designed with the idea of least privilege so that all principals have only the minimum number of permissions necessary to complete their tasks within the system, and no permissions to perform tasks outside of their allowed set.

Several different authorization schemes have been developed to achieve these various design principles; for example: mandatory or discretionary access controls, role-based or attribute-based access controls, etc. Each of these schemes conveys benefits and drawbacks; for example, role-based access control is often ideal when there are relatively few categories of principal that interact with the system but can become either difficult to manage or overly permissive when the roles become numerous. Often an authorization strategy is not a matter of choosing one specific approach but rather of determining which mixture of the various schemes together makes the most sense for an organization’s scenarios, to balance the benefits and drawbacks while still achieving the goals of the security design principles.

Organizations should codify their approaches for authorization so that they are implemented consistently within a product or service.

Establish Log Requirements and Audit Practices

In the event of a security-related incident, it is important to be able to piece together relevant details to determine what happened. Well-designed application, system and security log files provide the ability to understand an application's behavior and how it has been used at any moment in time. They are the fundamental data sources that inform automated Security Information and Event Management (SIEM) systems alerting.

The creation and maintenance of these logs is a critical set of software design decisions that is informed by the business and system requirements, the threat model, and the availability of log creation and maintenance functions in the deployed environment, with uses that range from security incident identification to system event optimization. The content of the log files should always be determined by the group or groups that will need to consume the log file contents. Because logging affects the available system resources of the local computer, it is important not only to capture the critical information but to restrict information capture to only the needed data. It is equally important to carefully identify what security information is relevant and needs to be logged, where the logs will be stored, for how long the logs will be retained and how the logs will be protected.

If possible, use the logging features of the operating system or other established tools, rather than creating a new logging infrastructure. The underlying infrastructure for platform logging technologies is likely to be secure and provide features such as tamper protection. It is critically important that any logging system provide controls to prevent tampering and offer basic configuration to ensure secure operation.

Ensure that security logs are configurable during application runtime. This allows the richness of logging information to be adjusted, which can help to provide more effective alerts to use in investigating security events.

Even with carefully considered and configured logging, it is very easy to accumulate vast amounts of log data, and therefore it is important to differentiate among logging needs. Monitoring data is relevant to configuration troubleshooting, usage, performance and ongoing feature development. Security logs are relevant to forensic analysis in the event of an incident. Unless logs are moved to a central location and archived, which is recommended, do not delete local log files too quickly. Deletion could potentially hinder required investigations into past events.

Secure Coding Practices

When developers write software, they can make mistakes. Left undetected, these mistakes can lead to unintentional vulnerabilities that potentially 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), using automated analysis tools and manually reviewing the code.

Establish Coding Standards and Conventions

When technology choices are made, it is important to understand which classes of security issues are inherited and to decide how they will be addressed. Once these technology decisions are 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. Where multiple options exist to address issues, choose one as the standard. Look for classes of security issues that a security feature, programming language or framework may mitigate on the developer's behalf and invest in the re-use of such features or frameworks instead of re-inventing them in-house. To the greatest possible extent, any frameworks/library/component use should be loosely coupled so that it can be easily replaced/upgraded when needed.

Standards must be realistic and enforceable. As coding standards and conventions are created, it is a great time to think about testing and validation. For example, what tools will you have at your disposal to help you validate that code follows the established policies? Would you need to rely on manual code review? Will it be possible to automate tests to help you with that validation? Incorporating the considerations above can lead to catching problems more effectively earlier in the SDL when they are less expensive to find and fix.

Resources

- OWASP – Secure Coding Practices, Quick Reference Guide
- Secure Coding Guidelines for Java SE
- Cert Secure Coding Wiki

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 string and buffer manipulation 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 described in the preceding section. Additionally, tooling should be deployed to assist in identifying and reviewing the usage of dangerous functions. Many static analysis and linting tools provide a mechanism to identify usage during build time, and some integrate into IDEs (Integrated Development Environments) to provide authoring time guidance to developers as they write code. 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.

CWE (Common Weakness Enumeration) References

- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CWE-805: Buffer Access with Incorrect Length Value

References

- ISO/IEC 9899:201x Programming Languages C - Annex-K Bounds Checking Interfaces
- Updated Field Experience with Annex K — Bounds Checking Interfaces
- Security Development Lifecycle (SDL) Banned Function Calls

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. Commonly, as languages evolve over time they incorporate security features, and developers using previous compiler and toolchain versions cannot make use of these security improvements in their software. For example, memory corruption issues, including buffer overruns and underruns, remain a common source of serious vulnerabilities in C and C++ code. To help address these vulnerabilities, it is important to use C and C++ compilers that 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.

Over time, software projects become dependent on a certain compiler version, and changing the version can be cumbersome; therefore, it is essential to consider the compiler version when starting a project.

CWE References

- CWE-691: Insufficient Control Flow Management
- CWE-908: Use of Uninitialized Resource

Verification

- A Microsoft tool named the BinSkim Binary Analyzer can verify whether most of the compiler and linker options (/GS, /DYNAMICBASE, /NXCOMPAT and /SAFESEH) are enabled in a Windows image: https://github.com/Microsoft/binskim.

Resources

- This article examines Microsoft and GCC toolchains for the C, C++ and Objective C languages and how they can be configured to provide a wide variety of security benefits: https://www.owasp.org/index.php/C-Based_Toolchain_Hardening.

Use Code Analysis Tools to Find Security Issues Early

Using tools to search the code to identify deviation from requirements helps verify that developers are following guidance and helps identify problems early in the development cycle. Build the execution of these tools into the "normal" compile/build cycle. This relieves developers of having to make special efforts to ensure that coding standards are being consistently followed. Using static analysis tools that plug directly into the IDE allows developers to find security bugs effectively without leaving their native IDE environment and is on par with using the latest versions of compilers and links with appropriate security switches, as discussed in the previous section.

Readers should also be aware that frameworks can only offer protection up to a certain point and developer-introduced bugs could easily lead to security vulnerabilities. Secure code review is a good way to identify vulnerabilities that result due to logic bugs in the source code.

This practice and its verification are covered in more detail in the Testing and Validation section.

Resources

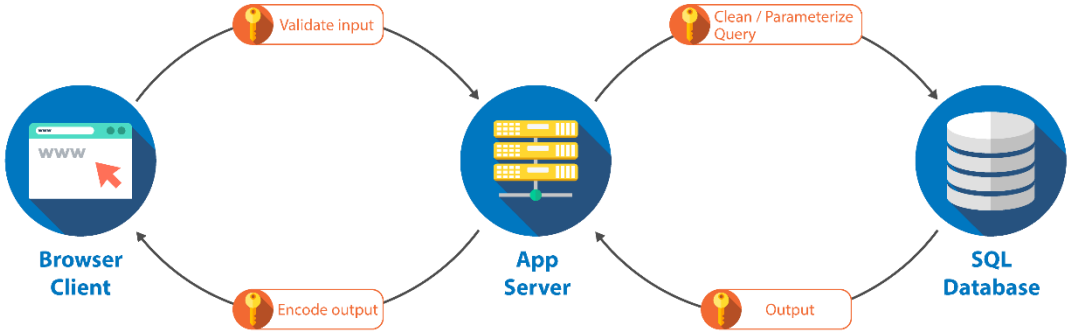
- A Microsoft tool named the DevSkim is a set of IDE plugins that provide inline analysis in the development environment as the developer writes code. The goal here is to give developers notification when they introduce a vulnerability (that can be detected) in order to fix the issue at the time of introduction and help educate the developer -- https://github.com/Microsoft/DevSkim.

Handle Data Safely

This is "Security 101:" all user-originated input should be treated as untrusted. The origin of data is often not clearly defined or understood, and applications can consume and process data that originates from the Internet or other network, through a file, from another application via some form of inter-process communication or other data channel. For example, in web applications the same user 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. If this data is not handled correctly at any point, it can, through a flaw in the application, be transformed into executing code or unintentionally provide access to resources.

In these situations, input validation is often identified as a defense that is applied at the boundary with the internet. This approach is inadequate, and a better approach is to ensure that each area in the application

stack defends itself against the malicious input it may be vulnerable to. For example, parameterized queries can be used for database access to help prevent SQL injection attacks. A good threat model will help identify the boundaries and the threats.



Each component protects against malicious input

Issues also arise when validation filters are not applied recursively or when using multiple steps in the validation logic. If more than one validation step is performed, take care not to provide an opportunity for an attacker to exploit the ordering of the steps to bypass the validation.

However, even when taking these steps, many vulnerabilities remain as a result of processing data in unsafe ways, and therefore input validation should only be considered as a defense in depth approach; other steps such as enforcing data segregation can help prevent data from becoming application logic. Such steps may include:

- Encoding, which ensures the data is transformed so that it will be interpreted purely as data in the context where it is being used
- Data binding, which prevents data from being interpreted as control logic by binding it to a specific data type

These strategies are not mutually exclusive, and in many situations both should be applied together. Even with all of these approaches, data validation is tricky to get right. And blocking known bad and allowing known good with some validation may be the only option in some instances, especially if there is no data segregation technique available for a specific context in which the data is used.

A different problem can occur with canonicalization. Data may be encoded into various formats to allow special characters and binary data to be used, and different components in the application stack may not always correctly parse or process those formats. This can lead to vulnerabilities. Canonicalization is the process for converting data that establishes how these various equivalent forms of data are resolved into a “standard,” “normal” or canonical form. Canonical representation ensures that the various forms of an expression do not bypass any security or filter mechanisms. When making security decisions on user supplied input, all decoding should be executed first using appropriate APIs until all encoding is resolved. Next, the input needs to be canonicalized, then validated. Input after canonicalization should be validated and either accepted or rejected. Only after canonicalization can a decision be made.

Note that sanitization (ensuring that data conforms to the requirements of the component that will consume it) is not the same as canonicalization. Sanitization can involve removing, replacing or encoding

unwanted characters or escaping characters. Input from untrusted sources should always be sanitized and when available the sanitization and validation methods provided by components should be used because custom-developed sanitization can often miss hidden complexities.

Canonicalization, Validation, and Sanitization can be easily confused, as they frequently are used together. As a simple example, many developers will at some point have to deal with the fact that different countries and cultures write the date in different ways. They recognize that "Jan 30, 2018," "30-1-2018," and "2018/01/30" are all different ways to represent the same date, and canonicalization is the process of translating those different representations into a consistent format (e.g., MM-DD-YYYY in the US) to avoid confusion. Sanitization, on the other hand, would be recognizing that "<script>alert(1)</script>" doesn't belong in a date and stripping it out. Validation would be recognizing that "<script>alert(1)</script>" doesn't belong in valid input for a date, and rather than stripping it out, rejecting the input. In that model, Validation replaces Sanitization, and is typically the safer approach. Validation would also involve verifying that Canonicalization of the date was successful, and that the final date was within whatever range the application expected.

Dates are one example where content may have different representations for the same information based on how different users chose to represent the data, but applications will additionally encounter different formats based on how computers might represent data. File paths, encoding schemes, character sets (8 Bit ASCII, 7 Bit ASCII, UTF-8, UTF-16, etc.), and little and big endian binary formats are all examples of differing data format schemes that applications may need to canonicalize. In many cases the formats are sufficiently complex that they require significant expertise to either Canonicalize, Validate, or Sanitize, and the most feasible approach is to use a library created by experts on the formats to perform one or more of those functions.

CWE References

- CWE-20: Improper Input Validation
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Verification

- OWASP offers pertinent testing advice to uncover SQL injection issues (see Resources, below). Various tools can help detect SQL injection vulnerabilities. Some of these are listed in the Tools section, below.
- Due to the variety of encodings and formats, verification of correct and complete canonicalization is a good candidate for fuzzing and code review. See "Perform Fuzz/Robustness Testing" in the Testing Recommendations section for more details.

References

- OWASP; SQL Injection; https://www.owasp.org/index.php/SQL_Injection
- OWASP; Data Validation; https://www.owasp.org/index.php/Data_Validation
- Writing Secure Code 2nd Ed.; Chapter 11 "Canonical Representation Issues;" Howard & Leblanc; Microsoft Press
- Hunting Security Bugs; Chapter 12 "Canonicalization Issues;" Gallagher, Jeffries & Lanauer; Microsoft Press

Books, Articles and Reports

- Preventing SQL Injections in ASP; Neerumalla; https://msdn.microsoft.com/en-us/library/cc676512.aspx
- MSDN Library; PathCanonicalize Function; https://msdn.microsoft.com/en-us/library/%20bb773569(VS.85).aspx
- Microsoft KnowledgeBase; How to Programmatically Test for Canonicalization Issues with ASP. NET; http://support.microsoft.com/kb/887459

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.

CWE References

- CWE 388: Error Handling
- CWE 544: Missing Standardized Error Handling Mechanism

Associated References

- https://www.owasp.org/index.php/Error_Handling

Manage Security Risk Inherent in the Use of Third-party Components

To innovate and deliver more value in shorter time periods, developers increasingly use third-party frameworks and libraries. These can be commercial off-the-shelf-products (COTS) or increasingly, these can come from open source software (OSS) projects. In either case, there is inherent security risk in the use of third-party components (TPCs) and these risks must be investigated and evaluated before use.

Often, these components are treated as black boxes and are afforded less scrutiny than comparable components written in-house. This different level of review introduces risks that, depending on security issues in the TPCs, may have an impact on the overall security of the systems in which they are used. Developers must be aware that they inherit security vulnerabilities of the components they incorporate and understand that choosing components for expedience (e.g., to reduce development time) can come at the cost of security when those components are integrated in production.

In general, you should choose established and proven frameworks and libraries that provide adequate security for your use cases, and which defend against identified threats. Do not waste resources and introduce new risks by re-implementing security features that are native to the framework. Without extensive testing, security failures will result and these security failures will often be nuanced and difficult to detect, leaving you and your users with a false sense of security.

Understanding the security posture is not enough to enable the making of informed decisions regarding a TPC's risks[3]. SAFECode's white paper "Managing Security Risks Inherent in the Use of Third-party Components" goes into detail on these challenges and offers guidelines for addressing them.

CWE References

- CWE-242: Use of Inherently Dangerous Function
- CWE-657: Violation of Secure Design Principles
- CWE-477: Use of Obsolete Functions

[3] The SAFECode Whitepaper "The Software Supply Chain Integrity Framework" provides a framework for assessing software supply chain integrity.

Testing and Validation

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.

Automated Testing

As in other areas of development, automated security testing 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. While some automated security tools can provide highly accurate and actionable results, automated security tools tend to have a higher rate of false positives and negatives than a skilled human security tester manually looking for vulnerabilities. Although some tools possess a large dictionary of flaws that is updated and expanded over time, currently they do not independently learn and evolve as they test, as a human does. Automated tools are adopted because of the speed and scale at which they run, and because 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. This section outlines six testing categories used by SAFECode members: 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.

Use Static Analysis Security Testing Tools

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. However, not every security problem lends itself to discovery via pattern or control flow analysis (e.g., vulnerabilities in business logic, issues introduced across multiple

application tiers or classes of issues created at runtime), nor are the rulesets used by SAST offerings typically completely comprehensive for all problems that hypothetically can be found via such analysis.

On-premise static analysis tools can benefit from human experts to tune the tool configuration and triage the results to make the results more actionable for developers and reduce false positive rates. Software as a Service (SaaS) solutions are available that are constantly tuned to reduce false positives and make results more effective. In either case, there is a compromise between false negatives (tuning out critical security issues) and false positives (non-issues being reported).

Perform Dynamic Analysis Security Testing

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.

If a DAST test is well authored it also provides validation that a flaw is actually exploitable. Most SAST tools identify patterns that are potentially exploitable if an attacker can control the data consumed by those patterns, but do not validate that an attacker can control that data. With DAST, if a test is successful, it has validated that an attacker actually can exploit the vulnerability.

However, DAST has limitations. It is much slower than SAST and can only test against functionality it can discover. Because DAST solutions can crawl a website enumerating functionality after some initial setup, web applications functionality discovery is semi-automatic. For other applications (desktop, mobile clients, specialized server applications, etc.) where there is not a standardized markup describing the interface, most DAST solutions require the interactions to be manually programmed. Once set up, the tests can be repeated with little additional cost, but there is an initial setup cost. Additionally, DAST solutions typically have a much smaller dictionary of scenarios they look for relative to SAST. This is because the cost for the tool maker to produce a reliable test is higher. DAST solutions often have difficulty scanning single-page applications (SPA) due to the fact that the page is not reloaded but is dynamically rewritten by the server. For organizations with SPA requirements, care should be taken in selecting a testing solution that meets the application needs. Coupling with SAST will offer fuller scanning coverage for all applications, and even more so for SPA.

Fuzz Parsers

Fuzzing is a specialized form of DAST (or in some limited instances, SAST). Fuzzing 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. While any single instance of a fuzzed input is unlikely to discover vulnerable behavior, repeating the process thousands or many millions of times (depending on how thoroughly the software had been fuzzed in the past) often will.

The benefit of fuzzing is that once the automation is set up it can run countless distinct tests against code with the only cost being compute time. Fuzzing produces far better coverage of parsing code than either traditional DAST or SAST. However, depending on scenario, it can be laborious to set up. Also, debugging the issues that fuzzing finds can often be time consuming. Fuzzing will have the biggest payoff for parsers written in non-memory-safe languages (e.g., C, C++) that can be easy for an attacker to access with malicious data. A threat model is quite helpful for identifying these parsers.

Network Vulnerability Scanning

This testing category is more prevalent in information and network security programs than SDL because vulnerability scanning tools find previously discovered vulnerabilities (CVEs), including those in third-party software, but are less useful for newly written applications. That said, these tools can add value to an SDL program as long as it is understood what vulnerabilities these tools are designed to identify. This category of testing is required for applications delivered as a physical or virtual appliance or deployed in a SaaS environment where security of the operating system environment is important. It is also valuable to scan any application in its installed state to ensure that no additional vulnerabilities are introduced during or after installation. To be most effective in this scenario, a baseline scan should be performed before the application installation, then the application should be scanned again after installation and configuration; comparing these results will give insight as to any vulnerabilities introduced by the application installation.

Verify Secure Configurations and Use of Platform Mitigations

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. Several tools can ensure that a compiled application has opted into OS platform mitigations such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), Control Flow Guard (CFG), and other mitigations to protect the application. Similarly, there are several security tools and freely available web services that verify that a website is opting into security-relevant HTTP headers such as x-frame-options, content security policy, HSTS, secure and HTTP Only cookie flags, and that a server's SSL/TLS configuration is free of protocol versions and cipher suites known to be insecure. 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.

Perform Automated Functional Testing of Security Features/Mitigations

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.

Manual Testing

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.

Perform Manual Verification of Security Features/Mitigations

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 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 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.

Perform Penetration Testing

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 (even companies with their own testers on staff may seek consultants to test highly specialized scenarios if those scenarios are outside the expertise of their staff testers). 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.

Verification

The existence of security testing can be verified by evaluating:

- Documented business risks or compliance requirements that provide prioritization for all testing activities (failed or missed test cases should be evaluated against these)
- Mitigating controls to identified threats, misuse (abuse) cases, or attacker stories as requirements
- Security test case descriptions
- Security test results

- Penetration testing or security assessment reports

Resources

- Exploiting Software; Hoglund and McGraw; Addison-Wesley, 2004
- The Art of Software Security Testing: Identifying Software Security Flaws; Wysopal, Nelson, Dai Zovi and Dusti; Addison-Wesley, 2006
- Open Source Security Testing Methodology Manual; ISECOM; http://www.isecom.org/
- Common Attack Pattern Enumeration and Classification; MITRE; http://capec.mitre.org/
- Software Security; Chapter 6, Software Penetration Testing; McGraw; Addison-Wesley, 2006

Manage Security Findings

One of the primary goals of an SDL program is to identify software design or implementation weaknesses that when exploited expose the application, environment or company to a level of risk.

Performing the secure development practices outlined in this document will aid in identifying these weaknesses. However, simply performing these activities is not sufficient. Action should be taken to correct the identified weaknesses to improve the overall security posture of the product.

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. For a finding to be remediated, the risk is completely eliminated, and to be mitigated, the risk is reduced but not completely eliminated. When the issue cannot be completely remediated, an assessment must be performed to determine whether the residual risk is acceptable. Residual risk may be radically different between products, depending on the products' usage and the sensitivity of data being processed or stored, and other factors such as brand damage, and revenue impact should also be considered. Once the residual risk is understood, a decision must be made whether the risk is acceptable, whether additional mitigations are required, or whether the issue must be fully remediated.

To ensure that action is taken, these findings should be recorded in a tracking (ideally an ADLM) system and made available to multiple teams in the organization. Teams who may need access to this information to make informed decisions regarding risk acceptance and product release include the development or operational teams who will remediate or mitigate the risk and the security team who will review and provide guidance to both development teams and the business owners. To enable easy discovery of these findings, they should also be identified as security issues, and to aid prioritization they should have a severity assigned. The tracking system should have the ability to produce a report of all unresolved (and resolved) security findings.

Define Severity

Clear definitions of severity are important to ensure that all SDL participants use the same language and have a consistent understanding of the security issue and its potential impact. To support this understanding, it is recommended to define criteria to categorize issues' severity. First consider implementing a severity scale (e.g., Critical, High, Medium and Low or a finer grained scale, Very High, High, Medium, Low, Very Low, Informational), and next define the criteria that contribute to each severity category. If detailed criteria are not available or known, a possible starting point is mapping severity levels to Common Vulnerability Scoring System (CVSS) thresholds (e.g., 10-8.5 = Critical, 8.4-7.0 = High, etc.). CVSS is primarily used for confirmed vulnerabilities but can also be used to prioritize SDL findings based on their complexity of exploitation and impact on the security properties of a system. Evangelizing the severity scale and definitions across the organization helps to ensure that everyone is speaking the same language when discussing security findings.

Risk Acceptance Process

When an issue cannot be completely resolved or fully mitigated, or can be only partially mitigated, a risk approval or mitigation request should be approved before the product is released. It is advisable to anticipate these situations ahead of time, create a structured format to communicate them and define process or workflow, including responsibilities of the roles involved. Ideally, the format and process will be instantiated in the ADLM system in use. There should be clear delineation of who submits, reviews and approves these requests. It is equally important to ensure that appropriately empowered individuals are assigned the responsibility to accept or reject risk. This should vary by the residual risk; e.g., Critical Risk to VP of Business Unit, Medium Risk to Engineering Manager.

A goal of the structured format is to drive consistency and help all involved consider critical factors when making a decision that affects residual risk. An example structure used by a SAFECode member is TSRV: Technique (T) captures the type of mitigation in effect using MITRE's Monster Mitigations, Specifics (S) documents the specific compensating control in effect, Remaining Risk (R) outlines the risk that the mitigation does not address, and Verification (V) explains how the mitigation's effectiveness was verified.

Acceptance of risk must be tracked and archived. The record of risk acceptance should include a severity rating, a remediation plan or expiration or re-review period for the exception and the area for re-review/validation (e.g., a function in code should be re-reviewed to ensure that it continues to sufficiently reduce the risk). In some cases, to ensure that required mitigating controls outside the application are implemented, it may be necessary to produce documentation, such as a Security Configuration Guide.

Vulnerability Response and Disclosure

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.

There are two ISO standards that provide detailed guidance on how to implement vulnerability response and disclosure processes within an organization. SAFECode members encourage others to use these industry-proven standards to the extent possible.

- ISO/IEC 29147 – 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 – Vulnerability handling processes (requires a fee to download). Gives guidance on how to internally process and resolve reports of potential vulnerabilities within an organization

Define Internal and External Policies

It is important to define and maintain a vulnerability response policy to state your company's intentions when investigating and remediating externally reported vulnerabilities. Typically, a vulnerability response and disclosure policy should consider guidelines for vulnerability response internal to your company and publicly or external to your company. The disclosure process will likely be different in the case when a vulnerability has been disclosed privately and in the case when the vulnerability has been publicly disclosed and/or actively exploited, and each scenario should be clearly defined and documented in the response policy.

Internal policy defines who is responsible in each stage of the vulnerability handling process and how they should handle information on potential and confirmed vulnerabilities.

External or public policy is primarily intended for external stakeholders, including vulnerability reporters or security researchers who report vulnerabilities, customers and potentially press or media contacts. The external policy sets expectations with external stakeholders about what they can expect when a potential vulnerability is found.

Define Roles and Responsibilities

Large organizations often establish dedicated Product Security Incident Response Teams (PSIRT) or incident response teams whose charter is to define and manage the vulnerability response and disclosure process. Smaller organizations will need to scale the size of this effort appropriately based on the size/scale of products and number of incidents. While the PSIRT members should understand all policies, guidelines and activities related to Vulnerability Response and Disclosure and be able to guide the software development team through the process, everyone involved in software development at the

organization and supporting functions such as customer service, legal, and public relations should understand their role and responsibilities as they relate to Vulnerability Response & Disclosure Process.

Roles and responsibilities of each stakeholder should be clearly documented in the internal vulnerability response policy.

The PSIRT Services Framework from FIRST (The Forum of Incident Response and Security Teams) provides good overview of the services covered by a PSIRT.

Ensure that Vulnerability Reporters Know Whom to Contact

To help facilitate the correct response to a potential security vulnerability, it is important that vulnerability reporters or security researchers, customers or other stakeholders know where, how, or to whom to report the vulnerability. Ideally, this contact or report intake location is easily discoverable; for example, on a company website. If you have a reporting process for researchers that is different from the process for customers, such information should be clearly called out.

Since information about vulnerabilities may be used to attack vulnerable products, sensitive vulnerability information should be communicated confidentially where possible. Secure communication methods such as encrypted email addresses and public keys should be clearly documented within the internal and external vulnerability response policies.

Manage Vulnerability Reporters

Vulnerabilities will be reported by vulnerability reporters (or security researchers) and by customers, and receipt of reports should be acknowledged by the PSIRT in a timely manner. At the time of receipt, an expectation of how the organization will respond should be set and if necessary, a request for additional information (to support triage) from the reporter should be made.

While the reported vulnerability is being investigated, a best practice amongst SAFECode members is to keep in contact with the reporter with the latest status so that they know you are taking the report seriously. Otherwise, they may publicize their findings before the vulnerability is addressed. ISO/IEC 29147 – Vulnerability disclosure provides more detailed guidance on this topic.

Monitor and Manage Third-party Component Vulnerabilities

Almost all organizations will use third-party components (TPCs) in software development. These TPCs include both open-source software (OSS) and commercial off-the-shelf (COTS) components, and development organizations inherit the security vulnerabilities of the components they incorporate. It is critical for organizations to have a process to monitor and manage vulnerabilities that are discovered in such components. See SAFECode's "Managing Security Risks Inherent in the Use of Third-Party Components" for more detailed guidance on this topic.

Fix the Vulnerability

The reported vulnerability should be immediately triaged by the team that owns the vulnerable code for validation and remediation, if appropriate. The severity of the vulnerability should be determined (see Manage Security Findings) to assist in prioritizing the fixes. Other considerations may help to determine the relative urgency of producing the fix, such as potential impact, likelihood of exploitation, and the scope of affected customers. Other factors that may affect response timelines are the component that is affected (for example, some components require longer QA cycles or can only be updated in a major release), and where the product is in the development cycle when the vulnerability is discovered. If a predictable schedule is used for releasing security updates, the details should be documented in the external vulnerability response policy to set expectations.

In addition to fixing the reported issue, the software development team should consider the nature of the defect. If it is likely that there are additional defects of the same class in the codebase, then it is a better to address the defect in those places as well as the one reported. Being proactive and systematic in mitigation helps avoid repeated submissions of the same vulnerability in other areas of the codebase. If the team develops software that is reused by other teams, then dependent teams need to be notified, and should update their software accordingly.

Identify Mitigating Factors or Workarounds

Security fixes should be fully tested by the software development team. Additionally, where possible, mitigating factors or workarounds, such as settings, configuration options, or general best practices, that could reduce the severity of exploitation of a vulnerability should be identified and tested by the development team. These mitigations and workarounds help users of the affected software defend against exploitation of the vulnerability before they are able to deploy any updates.

Vulnerability Disclosure

When available, the fix should be communicated to customers through security advisories, bulletins, or similar notification methods. Security advisories and bulletins should be released only once fixes are in place for all supported versions of the affected product(s).

Typically, advance notification is not provided to individual customers. This ensures that all customers are not exposed to malicious attacks while the fix is being developed and receive the correct information to remediate the vulnerability once the fix is available.

The disclosure process will be different when the vulnerability is being publicly disclosed and/or actively exploited: in this case, it may be desirable to release a public security advisory before a fix for the vulnerability is ready for release. Make sure this aspect of the process is clearly defined and documented in the internal vulnerability response policy.

Certain vulnerabilities may require multi-party coordination before they are publicly disclosed. Please see "Guidelines and Practices for Multi-Party Vulnerability Coordination and Disclosure" from FIRST for guidance and best practices for such cases.

Security advisories and bulletins must strike a balance between providing sufficient details so that customers can protect themselves and being so detailed that malicious parties could take advantage of the information in the advisory or bulletin and exploit it to the detriment of customers.

Security Advisories and Bulletins will typically include the following information where applicable:

- Products, applicable versions and platforms affected
- The severity rating/level for the vulnerability (see Manage Security Findings)
- Common Vulnerability Enumeration (CVE: http://cve.mitre.org) identifier for the vulnerability so that the information on the vulnerability can be shared across various vulnerability management capabilities (e.g., vulnerability scanners, repositories, and services)
- Brief description of the vulnerability and potential impact if exploited
- Fix details with update/workaround information
- Credit to the reporter for discovering the vulnerability and working on a coordinated disclosure (if applicable)

ISO/IEC 29147 -- Vulnerability disclosure provides guidance for security advisories and bulletins in more detail.

Secure Development Lifecycle Feedback

To prevent similar vulnerabilities from occurring in new or updated products, perform a root cause analysis and update the secure development lifecycle using information gained from the root cause analysis and during the remediation process. This feedback loop is a critical step for continuous improvement of the SDL program.

Resources

- Common Vulnerability Scoring System (CVSS) (https://www.first.org/cvss/): the industry standard vulnerability severity scoring system.
- Common Vulnerability Enumeration (CVE: http://cve.mitre.org): industry standard for uniquely identifying vulnerabilities to facilitate sharing information across vulnerability management tools such as vulnerability scanners, repositories and services.
- Forum of Incident Response and Security Teams (FIRST) (www.first.org): FIRST is an industry forum dedicated to best practices in incident response. FIRST has a variety of publications regarding best practices and training for incident response.
- CERT (http://cert.org/): A division of the Software Engineering Institute (SEI) that studies and solves problems with widespread cybersecurity implications.
- OWASP (https://www.owasp.org/index.php/SAMM_-_Vulnerability_Management_-_1): OWASP has a lightweight vulnerability response guide.

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

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. Learn from past successes and mistakes. 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.

Expertise and Skill Level of the Organization

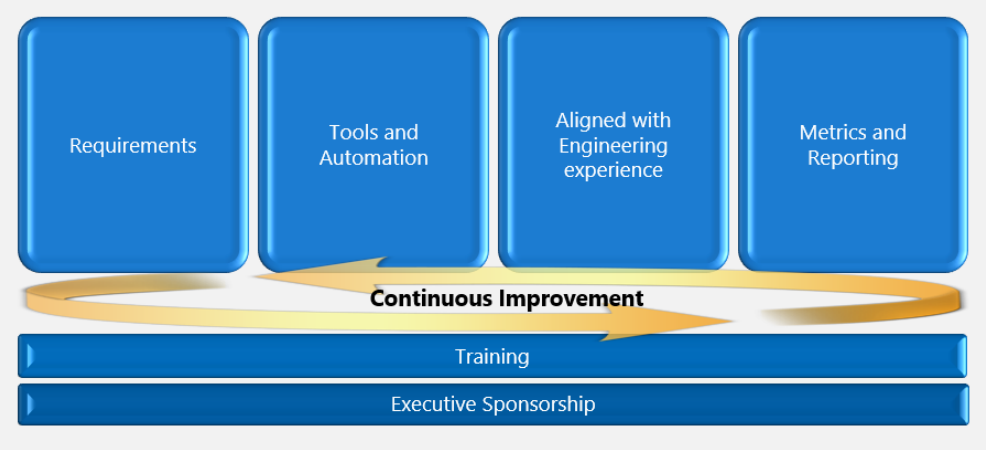
If an organization is to be successful in implementing an SDL, some level of training is necessary. The entire organization should be made aware of the importance of security, and more detailed technical training must be provided to development teams that clearly articulates the specific expectations 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. For example, the Secure Design Principles practice requires threat modeling expertise and possibly cryptography expertise. Does the organization have any resident experts in these areas? Is the number of

experts sufficient? Can these experts assist with training and establishing the expertise within the organization? SAFECode (https://training.safecode.org/) has a series of free, publicly available training courses to assist with establishing and building this expertise.

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.



SDL framework for all development models

The SDL framework should include guidance as to when a secure development practice applies and when it must be complete. This guidance must map to the development model and terminology used by development teams.

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? Depending on the maturity of the organization, it may be wise to start with a subset of the secure development practices and establish traction and expertise in those areas prior to full-scale implementation of all of the practices.
- 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 team is probably marching towards a tight timeline and has not included in its schedule the time to learn and adopt a new process or set of security requirements.
- 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. For example, an existing product whose development team is working on a new release may have an architecture whose threat model reveals vulnerabilities that are both very serious and very difficult to mitigate. It may make sense to agree with such a product team that they will “do what is feasible now” and transition to a new architecture over an agreed period of time.

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.

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? Ideally, the ADLM system and testing environments can be configured to produce this evidence "for free” as a side-effect of executing the secure development process.
- 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?

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.

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.

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.

To continue this positive trend, SAFECode encourages other software providers not only to consider, tailor and adopt the practices outlined in this paper, but also to continue to contribute to a broad industry dialogue on advancing secure software development. For its part, SAFECode will continue to review and update the practices in this paper based on the experiences of our members and the feedback from the industry and other experts. To this end, we encourage your comments and contributions, especially to the newly added work on verification methods. To comment on this paper, please write to feedback@safecode.org. To contribute, please visit www.safecode.org.

Acknowledgements

Authors

- Tony Rice, Microsoft
- Josh Brown-White, Microsoft
- Tania Skinner, Intel
- Nick Ozmore, Veracode
- Nazira Carlage, Dell EMC
- Wendy Poland, Adobe
- Eric Heitzman, Security Compass
- Danny Dhillon, Dell EMC

Contributors

- Edward Bonver, Symantec
- Anders Magnusson, CA Technologies
- Dermot Connell, Symantec
- Ravindra Rajaram, CA Technologies
- Jim Manico, Manicode Security
- John Martin, Boeing
- Manuel Ifland, Siemens
- Prithvi Bisht, Adobe
- Steve Lipner, SAFECode

Reviewers

- Linda Criddle, Intel
- Peter Chestna, CA Technologies
- Dave Lewis, Symantec
- Martin Baur, Siemens
- Sandro Kaden, Siemens
- Izar Tarandach, Autodesk

About SAFECode

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. Its charter members include Adobe Systems Incorporated, CA Technologies, EMC Corp., Intel Corp., Microsoft Corp., Symantec Corp. and Siemens. Associate members include Autodesk, Boeing, Cigital, Huawei, NetApp, Security Compass, Synopsis, Veracode and VMWare. For more information, please visit www.safecode.org.