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

Practical Security Stories and Security Tasks for Agile Development Environments

JULY 17, 2012

Table of Contents

Problem Statement and Target Audience

One of the strengths of the Agile development technique lies in it being an "early feedback system" that can potentially identify defects much earlier in the software lifecycle than conventional software development techniques. With Agile, necessary changes are incorporated in a dynamic fashion. Cycles/sprints are very short, usually no more than two to four weeks, and for this reason software development teams find it difficult (if not impossible) to comply with a long list of security assurance tasks. Consequently, more often than not, the development team ends up skipping security tasks altogether; shipping software with a high software security risk level.

This paper attempts to address this gap by providing Agile practitioners with a list of security-focused stories and security tasks they can consume "as is" in their Agile-based development environments. The objective of the paper is to go a step beyond providing a list of security flaws and translate secure development practices into a language and format that Agile practitioners can more readily act upon as part of a standard Agile methodology. To simplify things further, the recommended security tasks are broken down by roles, including architects, developers and testers, and the tasks that most often require specialized skills from security experts are listed separately. As such, this paper can readily serve as a first stop for organizations which:

1. Already use Agile methods and wish to incorporate security tasks into or enhance existing security tasks used in their development process.

2. Use the 'waterfall' development technique with adequate security measures already in place, but are evaluating moving to Agile methods without re-inventing the wheel.

3. Use a non-Agile development technique but need to interoperate with Agile environments, for example, a component development outsourced to a vendor using Agile development technique.

Overview

This paper consists of three Sections with content tailored specifically to the unique needs of Agile architects, developers and testers.

Section 1

- Provides an overview of a sample Agile methodology and a set of security tasks that may be beneficial.

Section 2

- **Section 2a** consists of **36** security-focused stories with associated security tasks. The "threat landscape" for this section was developed based on the most common issues SAFECode members are seeing in their own environments. In addition, the *CWE/SANS Top 25 Most Dangerous Development Errors* list (plus the 16 weaknesses on the cusp list) and the *OWASP Top 10* list were consulted for this section to ensure completeness of coverage. A list of unique "security-focused stories" was derived from this threat landscape, followed by associated common security tasks for the stories.
- **Section 2b** consists of a set of **17** operational security tasks that Agile practitioners should consider conducting on an ongoing basis. These have been further classified as Required or Recommended for new or existing code, or for the software development team in general.

Section 3

- Consists of **12** advanced security tasks that typically require guidance from software security experts (in-house or consultants) for the first

few iterations or in an ongoing manner. These tasks relate more to the competencies of the team members and their way of working.

Assumptions

1. The audience for this paper should already understand the fundamental nature of Agile software development. The key concepts and terms are derived from Scrum or Scrum-like methodology. However, the focus is on helping the reader implement a practical approach to building secure software via Agile concepts and not to promote any individual practice, as each organization will have different needs.

2. A team's architects, developers and testers were kept in perspective, instead of the end user, when choosing security-focused story names. While user stories center around "use cases" that allow the user to complete a certain task, security-focused stories revolve around "abuse cases," which don't reflect a typical end-user view of the system, and to which the end-user has no visibility or participation.

Section 1) Agile Development Methodologies and Security

In Agile, the business requirements are typically defined as user stories or epics (group of related user stories). These describe expected user scenarios ("use cases," "features," etc.) at a fairly high level. The focus is on defining system functionality with an affirmative approach ("as a user I want to access my private data") instead of by negation of an end-state or condition ("as a user I do not want my data to be exposed"). These user stories are then broken into more manageable and concrete tasks that the sprint team implements. They can be very detailed and technical in nature and therefore are prime candidates for defining the detailed security aspects of the system.

How to Choose the Security-focused Stories and Security Tasks?

1. An initial round of security analysis should be conducted to ensure that product management understands the security-relevant aspects of business requirements. These include envisioned system functionality, policy/compliance, legal, contractual and regulatory requirements, and in best cases, clear security-related requirements. For example, if the software handles and stores credit card holder data, it is likely subject to the PCI DSS requirements. At this time, product management can be consulted to verify they are prepared for this and understand the potentially strict compliance obligations associated with such a requirement.

   Product management should then work through these individual requirements and define more detailed backlog items that are then entered into the backlog along with a proper prioritization. At this time, appropriate security-focused stories from **Section 2a** should be considered to ensure user stories are covered by relevant security stories. Security stories that are purely non-functional, for example "as an architect/developer, I want to ensure graceful handling of all exceptions," need to be placed into the backlog as agreed with the Product Owner and the development team.

   When a new development sprint starts, the development team should pick up (or be assigned) the tasks allocated for the sprint (both functional and security-focused tasks) and commence the development work.

2. Security tasks listed in **Section 2b** should be incorporated in an ongoing manner in your Agile development life cycle.

3. Security tasks listed in **Section 3** should be incorporated as the tasks from **2a** and **2b** are accomplished.

Note: To maximize the effective use of this work, security tasks associated with the security-focused stories selected during sprint planning must be part of the version control pre-check-in task list for new/modified code (as applicable). The QA team should also create relevant test cases mapped to the security-focused stories and execute all of them during testing.

Story and Task Prioritization Using "Security Debt"

The term "security debt" in Agile software development is used to describe uncompleted tasks that have security relevance. Skipping, postponing, de-prioritizing or otherwise ignoring applicable security-focused stories or security tasks will build "debt" that by accumulating will likely leave the application vulnerable. Sometimes this is addressed by performing security sprints that try to clear or reduce this debt, but it is recommended to try to avoid building debt in the first place. Unfortunately, sometimes it is impossible to not build security debt. Prioritization of some kind is necessary in such cases. In order to accomplish that, take the following factors into account:

1. **Return on investment:** Consider some of the proposed stories and tasks; it will be easy to see that they directly influence not only the coding of a system, but its very design. If a team must concentrate on a select number of stories and tasks, then the ones in this paper might give the team the most return on investment; but if left unaddressed, these tasks are the ones that will cost the most to revisit once the system has been designed and implemented. For example, the completion and maintenance of a comprehensive threat model may be a costly activity, but one that will have long-term benefits due to the number of defects it can reveal if performed early, and which will make the tasks of system architecture and documentation easier in the future.
2. **Nature of your system:** Another factor when considering security debt is the nature of your system, its deployment, and the nature of the data for which it will be responsible. Given these factors, it may be apparent that one particular attack vector would be more readily prevalent than others, and that might influence the security stories and tasks you choose to address first. It does not mean that the others are less important, but simply that the conditions of your system dictate that these activities will minimize your system's attack surface versus others that might be "good to have" but do not have as immediate an impact on your system's security posture.

Residual Risk Acceptance

It is important to both inform business stakeholders of any known risks left in the application that is approved for release to production and get their approval for leaving these risks unresolved. The sprint review conducted at the end of a given sprint should cover any issues left in the completed sprint. A single release may include a large number of sprints performed by various teams, and thus there can be multiple known risks and issues in a release. Any risk that may impact the business objectives must be approved by the respective stakeholders before release. Undone work must go into the backlog to ensure it is tracked. Refer to **Appendix A** for more details.

Section 2a) Security-focused Stories and Associated Security Tasks

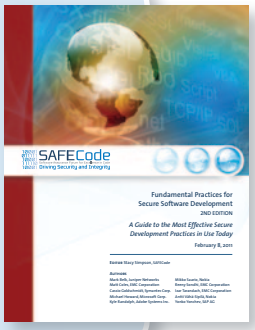
The purpose of this section is to provide an actionable list of security tasks that Agile architects, developers and testers can perform according to their specific roles to ensure that security considerations are addressed throughout the development process.

While SAFECode's *Fundamental Practices for Secure Software Development* already lists a set of engineering tasks for creating more secure software, it may not be readily apparent to Agile development teams how best to incorporate these tasks into their unique environments. This section breaks down the Fundamental Practices into familiar Agile "stories" focused on security and derived from the issues most commonly seen by SAFECode members in their environments. Both the *CWE/SANS Top 25 Most Dangerous Development Errors* list (plus the 16 weaknesses on the cusp list) and the *OWASP Top 10* list were also consulted to ensure broad coverage.

In addition, each security-focused story has been associated with a list of corresponding backlog tasks, which are based on the observations the authors (or their internal peers working in the software security domain) have made while working with different software development teams within their respective organizations. These tasks were also reviewed by representatives from across SAFECode's membership to ensure their practicality and broad applicability. This mapping of Fundamental Practices to security stories to backlog tasks provides secure engineering guidance to Agile teams in a format that is familiar to them and that they can easily consume.

Fundamental Practices for Secure Software Development 2nd Edition: A Guide to the Most Effective Secure Development Practices in Use Today

www.safecode.org/publications/SAFECode_Dev_Practices0211.pdf



A few security tasks require combined actions from both Architect/Developer and QA resources. These have been prefixed as **A/D/T** (Architect/Developer/Test). Others have been tagged as **A/D** (Architect/Developer) or **T** (Test).

Corresponding Common Weakness Enumerations (CWE-ID) have also been indicated for those interested in understanding the underlying security weakness and its implications in detail.

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
1	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify allocation of resources within limits or throttling	**[A]** Clearly identify resources. A few examples: • Number of simultaneous connections to an application on a web server from same user or from different users • File size that can be uploaded • Maximum number of files that can be uploaded to a file system folder **[A/D]** Define limits on resource allocation. **[T]** Conduct performance/stress testing to ensure that the numbers chosen are realistic (i.e. backed by data). **[A/D/T]** Define and test system behavior for correctness when limits are exceeded. A few examples: • Rejecting new connection requests • Preventing simultaneous connection requests from the same user/IP, etc. • Preventing users from uploading files greater than a specific size, e.g., 2 MB • Archiving data in file upload folder when a specific limit is reached to prevent file system exhaustion	• Validate Input and Output to Mitigate Common Vulnerabilities • Perform Fuzz/Robustness Testing	CWE-770

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
2	As a(n) architect/ developer, I want to ensure **AND** as QA, I want to verify application of appropriate encoding for output context	**[A]** Clearly identify all types of output context. A few examples: • Output is rendered only as HTML • Output is rendered as HTML attributes • Output is rendered as a URL **[D]** Adhere to SAFECode's Fundamental Practices for Secure Software Development for proper encoding of output context. Prefer use of language-specific in-built APIs such as HTMLEncode() (for C#) for encoding purpose. If that's not feasible, use well-known encoding frameworks/controls such as ESAPI encoder. Also, do canonicalization of user input to prevent bypass of encoding filters that have been applied. **[T]** Use a combination of manual test cases and automated means (web vulnerability scanners) in order to verify the strength of encoding filter applied.	• Use Anti-Cross Site Scripting (XSS) Libraries • Validate Input and Output to Mitigate Common Vulnerabilities	CWE-838
3	As a(n) architect/ developer, I want to ensure **AND** as QA, I want to verify application of or access within index boundaries of buffers and arrays	**[A/D]** Define where buffer operations (on dynamic buffers) occur. Define data types and bounds for buffer operations. **[D]** Adhere to SAFECode's Fundamental Practices for Secure Software Development for prevention of buffer overflows. **[D]** Scan source code for such violations using static code analyzer tools, e.g., Coverity. **[A/D]** Conduct false positive analysis of flagged issues. **[D]** Fix buffer overflow issues analyzed as confirmed. **[T]** Use fuzz testing tools to verify that no process/system crashes/hangs exist. If they do, fix them and re-run the tool.	• Minimize Use of Unsafe String and Buffer Functions • Use a Current Compiler Toolset • Use Static Analysis Tools	CWE-120 CWE-131 CWE-805

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
4	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify graceful handling of all exceptions	**[A/D]** Application should have provisions to catch all exceptions. **[A/D]** Application exception codes should be clearly defined. **[A/D/T]** Unknown exceptions should be tied to a generic error code. **[A/D/T]** Any exception condition in the application should not throw stack trace (or similar information) to the end-user.	• Perform Fuzz/Robustness Testing • Use a Current Compiler Toolset • Use Static Analysis Tools	CWE-754
5	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify concurrent execution using shared resources with proper synchronization	**[D]** Scan source code for race condition violations using static code analyzer tools, e.g., Coverity. **[A/D]** Conduct false positive analysis of flagged issues. **[D]** Fix race condition issues analyzed as confirmed. **[T]** Use fuzz testing tools to verify that process/system crashes/hangs don't exist. If they do, get them fixed and re-run to verify.	• Perform Fuzz/Robustness Testing • Use Static Analysis Tools	CWE-362
6	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify use of controlled format string	**[D]** Adhere to SAFECode's Fundamental Practices for Secure Software Development for preventing format string issues. **[D]** Scan source code for such violations using code analyzer tools, e.g., Coverity. **[A/D]** Conduct false positive analysis of flagged issues. **[D]** Fix format string issues analyzed as confirmed. **[T]** Use fuzz testing tool to verify that no process/system crashes/hangs exist. If they do, fix them and re-run the tool.	• Minimize Use of Unsafe String and Buffer Functions • Use Canonical Data Formats • Use Static Analysis Tools • Perform Fuzz/Robustness Testing	CWE-134

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
7	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify use of controlled integer bounds	**[D]** Adhere to SAFECode's Fundamental Practices for Secure Software Development for preventing integer overflows. **[D]** Scan source code for such violations using code analyzer tools, e.g., Coverity. **[A/D]** Conduct false positive analysis of flagged issues. **[D]** Fix integer overflow issues analyzed as confirmed. **[T]** Use fuzz testing tools to verify that no process/system crashes/hangs exist. If they do, fix them and re-run the tool.	• Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets • Use Static Analysis Tools • Perform Fuzz/Robustness Testing	CWE-190
8	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that users have access to the specific resources they require which they are authorized to use	**[A]** Create a detailed authorization matrix that specifies which user groups/users have access to which resources (folders, files, UI, etc.). **[D]** Ensure that your application's authorization mechanism complies with the matrix created in step (for example, if role-based access control [RBAC] is used, ensure it corresponds to the authorization matrix created). **[T]** Test effectiveness by using a combination of manual and automated means.	• Use Least Privilege	CWE-862 CWE-863
9	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify correct conversion between numeric types	**[D]** Adhere to SAFECode's Fundamental Practices for Secure Software Development for preventing type conversion errors. **[D]** Scan source code for such violations using code analyzer tools, e.g., Coverity. **[A/D]** Conduct false positive analysis of flagged issues. **[D]** Fix incorrect numeric type issues analyzed as confirmed. **[T]** Use fuzz testing tool to verify that no process/system crashes/hangs exist. If they do, fix them and re-run the tool.	• Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets • Use Static Analysis Tools • Perform Fuzz/Robustness Testing	CWE-681

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
10	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify correct permission assignment and maintenance for all critical resources	**[D/T]** When a critical resource is defined or accessed, make sure that the access permissions (programmatic and systemic) to it are left in their most restrictive but useful possible setting. **[D]** Describe correct permissions for the resource in the security configuration guide.	• Use Least Privilege	CWE-732
11	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that sensitive data is kept restricted to actors authorized to access it	[**A/D/T]** When sensitive data (either user data that's considered sensitive or system data that may lead to insecure outcomes if leaked) is transferred by any channels across a trust boundary (for example, internal IP addresses as part of an HTTP or SMTP header, or a full internal filename with path is exposed in a GUI), be sure to remove the sensitive part. **[A]** Clearly specify which data produced by the system is to be considered sensitive and socialize that status across the development team and QA. **[D/T]** When handling sensitive data, have your code fail gracefully so that sensitive data does not leak. **[D/T]** Make sure that sensitive information is not leaked in error messages and stack traces.	• Use Least Privilege	CWE-212
12	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the same steps are followed in the same order to perform an action, without possible deviation on purpose or not	**[A/D/T]** When creating and verifying the business logic of multiple-step actions in the system, ensure that the action cannot suffer from missing steps, that steps cannot be performed in an arbitrary order, and that there is a timeout in each step that invalidates the whole operation. **[D/T]** In case of timeout or user-cancelation of the action, be sure that all initiated changes are rolled back to the state they were in before the action started. **[D/T]** Be sure that all database commits and system state changes are only affected after the business logic has been validated and the action has been completed.	• Threat Modeling	CWE-841

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
13	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the damage incurred to the system and its data is limited if an unauthorized actor is able to take control of a process or otherwise influence its behavior in unpredicted ways	**[A]** Make sure distinct processes that need to communicate can do so in a way that only requires the minimum set of permissions (for example, don't run two processes as root just because they both need to access the same resource in a Unix environment). **[D]** Make sure any process that needs to have privileged access has it only for the minimum amount of time necessary and is able to drop the privileges as soon as they are not needed (for example, a network service opening a port lower than 1024 and dropping the elevated privileges necessary to do so right after the port is successfully opened). **[D]** Make sure that privileges are dropped correctly as part of privileged operations exception handling. **[A/D/T]** Make use of operating systems facilities like dedicated users, operating system capabilities matrices, jails and sandboxes to limit the exposure of the system to exploitation of a given process. **[T]** Make sure when testing that the system can operate in a security-hardened environment (more restrictive in terms of privilege handling).	• Implement Sandboxing • Threat Modeling • Determine Attack Surface	CWE-250

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
14	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that every resource and system that I access as part of operating my own system is providing me with verified services and content, and that content I provide gives my customer protection and verification against substitution and tampering in-transit	**[D/T]** When relying on the content of remote systems for the correct operation of your system, verify both the identity of the remote system (by using DNS and reverse-DNS queries and/or SSL certificates) and the integrity and authenticity of the content acquired (by using signatures and hashes). **[A/D/T]** Make use of code-signing technologies like Authenticode, jar signing, etc., as appropriate per content when consuming, and provide when producing. **[D/T]** Make sure to perform all necessary checks to validate the object being checked depending on the technology used (certificate chain of trust, for example). **[A/D]** Make proper use of cryptography tools to ensure integrity of a given value between its inception and its use. **[A/D]** Store all sensitive information used for security decisions on the server only—do not rely on client-side security decisions. **[A/D]** Use session management frameworks over stateless protocols to maintain security-decision values. **[A/D/T]** Understand the data flows of your application and identify those where information used for security decisions can be intercepted and tampered with and armor those channels.	• Eliminate Weak Cryptography • Threat Modeling	CWE-494

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
15	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the system does not import functionality from sources that are not under the system's control	**[D]** Do not include the capability to refer to code that is defined, stored and controlled in an external location outside the control mechanisms of your system and that can interact with the system and its components. [**D/T**] If you absolutely must import functionality from external sources, make sure that relevant server-side checks are applied to all content generated by the imported code. Do not trust the imported functionality "as is." **[A]** Use an application-level firewall that can guard against this kind of vulnerability.	• Threat Modeling	CWE-829
16	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify limitation of a pathname to a restricted directory ('Path Traversal')	**[D/T]** When accepting external input that will be used to construct a file path, remove or invalidate special constructs like ".", "..", "\" and "/", including their many representations in alternate character sets. **[D/T]** Filter data repeatedly until all findings are exhausted to prevent nested constructs. **[D/T]** Only after cleaning up the input for extraneous characters, make sure that the full path received falls inside of the permitted area by using a whitelist of possible path locations. **[D]** Perform checks as close to the operation as possible, as content may change in transit. **[D]** Use language or operating system-provided functions to create a canonical form of the path and check it against your whitelist. **[A/D]** Prefer mappings and indexed menus instead of free-form input when choosing paths. **[D]** Examine relevant findings from static code analysis tools.	• Use Canonical Data Formats • Validate Input and Output to Mitigate Common Vulnerabilities	CWE-22

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
17	As a(n) architect/ developer, I want to ensure **AND** as QA, I want to verify that cross-site scripting attacks are prevented	**[D]** Consider all input as malicious and filter according to the context. **[D/T]** When generating dynamic web pages, filter the input for any browser-executable content that is not intended (for example, from user-originated fields in a database). Consider all forms of input of content that might eventually be presented to and consumed by a browser, like events generated outside the system, log messages, arguments in a URL, form field values, etc. Perform this filtering at server-side, close to use. **[D]** When generating dynamic web pages, encode the output to the needed character set and explicitly declare it as part of the page. **[D]** When generating dynamic web pages, sanitize the output by properly escaping and quoting the dynamic content in a way to properly enforce separation of code and data according to the environment in use. **[D]** Use automated scanning tools in a credentialed mode with maximum coverage of the application interface to test for this vulnerability. **[D]** Use one of the many available libraries that takes cross-site scripting into account; create and enforce a single way of filtering input for cross-site scripting injection. **[D]** Always use cookies (authentication/session) with HttpOnly attribute.	• Use Anti-Cross Site Scripting (XSS) Libraries • Validate Input and Output to Mitigate Common Vulnerabilities	CWE-79

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
18	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that cross-site request forgery attacks are prevented	**[D]** Use one of the many available libraries and frameworks that takes CSRF into account. **[D]** Defend against cross-site scripting (see Story 17). **[A/D]** Add business logic and workflow steps to critical processes in the system, and make them out-of-band: send an email in case of password change, send a text message when changing a critical value. **[D/T]** Log critical operations and the details of their initiation and arguments. **[A/D]** Do not use HTTP GET for any method that effects a change in system state.	• Use Anti-Cross Site Scripting (XSS) Libraries • Validate Input and Output to Mitigate Common Vulnerabilities • Use Logging and Tracing	CWE-352
19	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify proper neutralization of Special Elements used in an OS Command ('OS Command Injection')	**[D]** Consider all input as malicious and filter according to the context. **[D]** Check all arguments to functions like exec() or system() for the expected format before executing. **[D]** Limit the use of external processes; prefer library calls. **[D]** Use static code analysis tools. **[D]** Consider the use of command shells [system()] as opposed to directly calling an executable [exec()] and its implications in command line arguments, like shell expansion. **[A/D]** Reduce the attack surface by adopting the backlog items of "Execution with Unnecessary Privileges."	• Validate Input and Output to Mitigate Common Vulnerabilities • Use Static Analysis Tools • Use Least Privilege	CWE-78

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
20	As an architect/ developer I want to ensure **AND** as QA I want to verify that database queries function as expected by separating the data from the query	**[D]** Follow best practices defined in SAFECode's Fundamental Practices for Secure Software Development: "Avoid String Concatenation for Dynamic SQL Statements." **[A/D]** Utilize common frameworks or libraries (such as OWASP ESAPI) that provide a secure database query functionality, as defined below. **[A/D]** Use prepared statements with bind variables (parameterized queries) that automatically enforce the separation between data and code. **[A/D]** Deploy the database user accounts with minimal access rights (least privilege) required to perform the database action. Use separate accounts for different access roles (read only, read and update, etc.). **[A/D]** Validate all input to ensure only allowed (whitelisted) set of characters is processed. **[A/D]** If dynamic SQL or stored procedures with user-supplied data is required, escape all parameters carefully using a database-specific escaping routine. **[A/D/T]** Comparable techniques apply also to XPath, NoSQL and other database queries. **[T]** Test all database queries created or used by the application to ensure they conform to the actual intent and structure, and cannot be manipulated by user input. **[T]** Utilize common SQL injection payloads and static/dynamic code analysis to ensure database access works as designed. **[T]** Ensure only pre-defined set of characters (whitelist) is processed by the system.	• Avoid String Concatenation for Dynamic SQL Statements • Validate Input and Output to Mitigate Common Vulnerabilities • Use Least Privilege	CWE-89

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
21	As an architect/ developer I do not want to store **AND** as QA I want to verify that the system does not store hard-coded sensitive information	**[A/D]** Store all sensitive credentials outside of the code in an encrypted, access-restricted configuration file or database accessible to a very limited number of users. **[A/D]** If possible, use hashes or keys instead of passwords. **[A/D]** Develop the application so that the credentials can be changed regularly. **[A/D]** All access to the credentials shall be logged on a separate storage. **[T]** Verify the credentials are protected and access is logged. **[T]** Apply black box methods, system-call tracing, and static/dynamic analysis to detect hard-coding weaknesses.	• Use Least Privilege • Eliminate Weak Cryptography • Use Static Analysis Tools	CWE-798
22	As a(n) architect/ developer, I want to ensure **AND** as QA, I want to verify that pointer-related checks are in place	**[D]** Check the results of all functions that return a pointer value and verify that the value is valid (e.g., not NULL and in range). **[T]** Use testing methods such as fuzzing and automated static code analysis tools to detect this flaw.	• Perform Fuzz/ Robustness Testing	CWE-822 CWE-825 CWE-476

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
23	As an architect/ developer I want to prevent **AND** as QA I want to verify there is no information exposure through error messages	**[A/D]** Ensure error messages only contain the minimal understandable details the user needs to know about the error. Do not return and display details such as stack traces, path names, or database query details in the response. **[A/D]** Do not use the client to hide server-side error details. The client should not make any other changes to the message other than apply formatting (style). **[T]** Systematically cause both in-the-system and application errors, and verify only approved information is returned and displayed back to the user. Ensure that not only web server errors (e.g., 404 page not found) are generic, but also that errors returned from the backend, such as the application or database server, do not contain any sensitive information (e.g., stack traces). **[T]** Use techniques such as fuzzing, static code analysis and fault injection to cause errors.	• Validate Input and Output to Mitigate Common Vulnerabilities	CWE-209

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
24	As an architect/ developer I want to ensure **AND** as QA I want to verify URL redirection to un-trusted sites is not possible	**[A]** Define a strict whitelist of accepted redirection destinations. If this is not possible, ensure only valid URLs are accepted. **[A]** Deny access to all other destinations. **[A]** Consider whether the user should separately be warned/notified about the redirection ("Leaving our site"). **[A]** Consider verifying on the server side that the destination URL shall not redirect the user to a different destination. While sometimes this may serve a legitimate purpose, it is often used to fool the user, e.g., by using URL shortening services or open redirectors on other sites to hide the real destination. Follow all detected redirects (up to a predefined count) and display a warning if any/too many are detected. **[A/D]** If possible, use mapping to ensure the destination URL is retrieved from a safe repository, such as having "destination=123" to correspond with a certain predefined URL. **[A/D]** If displaying the URL back to the user, ensure it is sanitized via input validation and cross-site scripting prevention mechanisms (see Story 17), as defined in this document and other common best practices. **[T]** Verify redirection can only take the user to approved destinations. **[T]** If there is no approved list, ensure no cross-site scripting attacks can take place (see Story 17) by testing with malicious URLs (e.g., containing JavaScript or malformed payload).	• Validate Input and Output to Mitigate Common Vulnerabilities • Use Anti-Cross Site Scripting (XSS) Libraries	CWE-601

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
25	As an architect/ developer I want to release a resource **AND** as QA I want to ensure resources are released after effective lifetime	**[A]** If possible, use a language that automatically handles garbage collection for de-allocated objects. **[D]** Consistently free all resources that have been reserved after they are no longer needed. **[D/T]** Verify that any failure in resource allocation places the system into a safe and recoverable posture and all throttling mechanisms function as intended. **[D]** Deploy built-in resource allocation limits in frameworks and platforms. [**T**] Test various parts of the system to ensure it is robust throughout. **[T]** Use methodologies such as static code analysis, load testing and fuzzing to identify unreleased resources.	• Use a Current Compiler Toolset • Use Static Analysis Tools • Perform Fuzz/ Robustness Testing	CWE-772
26	As a(n) architect/ developer, I want to ensure **AND** as QA, I want to verify that resources are initialized where necessary	**[A]** Consider using languages/frameworks that avoid these issues. [**A/D**] Ensure variables are properly initialized, especially when utilizing data from un-trusted sources. **[T]** Search for improperly initialized resources causing unusual error conditions in the system, using, for example, a static analysis tool, stress-testing, fuzzing, fault injection, and/or appropriate compiler settings.	• Use a Current Compiler Toolset • Use Static Analysis Tools • Perform Fuzz/ Robustness Testing	CWE-456

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
27	As an architect/ developer I want to prevent **AND** as QA I want to verify controls against unauthorized access to user accounts by password guessing	**[A]** Ensure access restrictions are imposed after a predetermined number of unsuccessful login attempts. **[A/D]** Prompt the offending user with additional challenges, such as CAPTCHA or other intensive tasks before allowing to try again. If the attempts continue, shorten the cycle and increase the computational cost and complexity. **[A/D]** Increase the subsequent response times to slow down the attack. **[A/D]** Raise an alarm to the system administration team. **[A/D]** Lock the targeted account. **[A/D]** Potentially release the account lock after a defined time, or require further action to re-enable the account. **[A/D]** Disable access by the violating user at the appropriate level: IP blocking (may cause denial of service to legitimate users), page redirection or session termination. **[A/D]** Log the failed login attempt, account locking and reopening of an account with sufficient detail. **[A]** Consider also alternative misuse cases, e.g., where a single password is tested against multiple accounts. **[T]** Verify the implemented controls function as designed. Think of/test new ways of bypassing them.	• Use Logging and Tracing • Threat Modeling	CWE-307

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
28	As an architect/ developer I want to ensure **AND** as QA I want to verify the user is protected by robust authentication and session management	**[A]** Define which areas of the application require authentication and authorization. **[A]** Utilize, as much as possible, common, robust authentication and session management solutions provided by platforms or frameworks. **[A/D]** Ensure credentials and sensitive session information is always transported over a secure channel such as the latest available version of TLS. **[A/D]** Prevent guessing or testing for existing usernames. If you must have this functionality, impose, e.g., throttling limits to prevent mass-harvesting of usernames. **[A/D]** Ensure, in cases of failed authentication attempts, the information returned to the user does not give away sensitive information such as whether the user account exists or not. **[A/D]** Ensure no channels exist to the system that have a weaker protection level than the rest of the channels. For example, if a service can be accessed from both the web browser and a native mobile application, they both should utilize a similar level of authentication and session management. **[A/D]** Ensure password or username brute force protection is built in as specified in Story 27 or other common best practices. **[A/D]** Ensure authentication and session management is enforced at all times (where needed). **[A/D]** Impose session expiration. **[A/D]** Follow best practices (e.g., OWASP Session Management Cheat Sheet) to prevent session management attacks. **[A]** Provide users convenient access to logout. ***continued on next page***	• Use Least Privilege • Threat Modeling	CWE-306

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
		continued from previous page **[A]** Ensure sessions are terminated on the server side once they expire or the user logs out. **[A]** Provide a secure method for recovering forgotten usernames or passwords. Do not display whether a particular username exists in the system. **[A]** If using email as username, consider providing a nickname in cases where the username is publicly used to identify the user. Examples are discussion forums or software feedback/ratings pages. **[A]** Utilize CAPTCHA or similar complex methods to slow down repeated attack attempts. **[A/D]** For sensitive transactions, consider requiring re-authentication. **[A/D]** For administrative functionality, consider using strong (multi-factor) authentication and separate, private channels. Consider providing security-conscious normal users a stronger authentication mechanism utilizing, e.g., another channel such as text messages. **[A/D]** Consider preventing the browser to cache, e.g., the password and/or username. **[A/D]** If providing a "remember me" functionality, make that optional for the user (opt-in) and use a separate secure cookie. Use expiration for this option that balances security and user convenience. **[T]** Test all of the defined features by using static code analysis, manual methods (complemented with the help from security experts), and web application scanners. For more complete explanation of issues and test cases, please refer, e.g., to OWASP's Testing Project, Authentication Cheat Sheet and Session Management Cheat Sheet.		

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
29	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that strong encryption is used for sensitive information	**[A]** Identify data that should be classified as "sensitive." Some examples of sensitive data include (but are not limited to): • Login credentials and tokens • Cryptographic private keys • Financial data such as credit card or bank account numbers, balances, or loan applications • Personal health data and medical records • Sensitive personally identifiable information (PII) including government identification numbers (such as Social Security numbers) **[A/D/T]** Use secure channels (such as SSL/TLS or IPsec) to transmit sensitive data across trust boundaries. **[A/D/T]** Encrypt sensitive data when "at rest," i.e., when persisted to a file or other data store. **[A/D/T]** Control access to decryption keys. **[A/D]** Do not hardcode keys into application code. **[A/D/T]** Use strong cryptographic algorithms to encrypt data, use cryptographically strong random number generators to generate initialization vectors for encryption routines. **[A/D]** Use strong key derivation functions (such as PBKDF) when generating encryption keys from user-provided passwords. **[A/D/T]** Tokenize sensitive data whenever possible.	• Eliminate Weak Cryptography	CWE-311

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
30	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that sufficient transport layer protection exists	**[A]** Always use secure channels (such as SSL/TLS) to transmit authentication credentials. Always use secure channels to transmit authentication tokens, including HTTP authentication cookies. All resources served to an authenticated user should be performed over a secure channel. **[A/D/T]** As a defense-in-depth measure, apply the "secure" attribute to HTTP authentication cookies to help prevent them from being sent over non-SSL/TLS connections.	• Eliminate Weak Cryptography • Threat Modeling	CWE-759
31	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that hashing uses a random salt	**[A/D/T]** When designing code to compare stored hashes to computed hashes (for example, when verifying a user's password in an authentication attempt), always store and append a per-hash unique, random salt value in order to hamper rainbow table attacks.	• Eliminate Weak Cryptography	CWE-327
32	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the cryptographic algorithm used is not broken or risky	**[A]** Always use vetted, industry-standard cryptographic algorithms to protect data. Do not attempt to develop your own algorithms. **[A]** Avoid using cryptographic algorithms with known weaknesses (such as MD) or that are beginning to show weakness (such as SHA-), whenever possible. **[A/D]** Take advantage of any cryptographic agility features supported by your application platform and language. **[A/D]** Avoid hard-coding particular algorithms into the application code. **[A/D]** Use abstract algorithm types and instantiate them from configuration files/stores.	• Eliminate Weak Cryptography	CWE-330

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
33	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify use of sufficiently random values to prevent sensitive information from being seen by unauthorized people	**[A/D/T]** Use cryptographically-strong random number generators whenever you need to protect a resource by making its name, identifier, or other property difficult to guess. Some examples of this include: • Session or authentication identifier tokens • Initialization vectors for cryptographic keys • Temporary file or directory names • Temporary or initial user passwords **[A]** Consider using a hardware-based random number generator for situations where randomness is critical.	• Eliminate Weak Cryptography	CWE-129
34	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify proper validation of array index	**[A]** When possible, prefer developing in a language with automatic array boundary checking. **[A/D/T]** For languages without automatic array boundary checking (such as C/C++): • Periodically test the application with a static analysis tool designed to detect potential array index violations. Ideally this testing would be performed against the source code repository either on a daily or as-checked-in basis. • Periodically test the application interfaces (including file and network parsing code) with a fuzz testing tool; investigate any crashes the fuzzer generates. • If the language supports a function parameter annotation language, use it to both clarify parameter meaning for human readers and to assist static analysis tools. • Compile and link your application with available automatic memory protection options such as address space layout randomization (ASLR) and NX (No eXecute)-bit support.	• Use a Current Compiler Toolset • Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets • Use Static Analysis Tools • Perform Fuzz/Robustness Testing	CWE-434

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
35	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the system does not permit unrestricted upload of files with dangerous type	**[A]** Ensure antivirus software is deployed to test all user-uploaded files. **[A/D/T]** Avoid storing user-supplied files in file form on web servers; instead, store them as binary objects in a data store. • When this design choice is infeasible, store the user-supplied files in a directory structure outside of the application webroot so that users cannot directly request them. • When possible, change the name of the uploaded file to a random value such as a GUID. Do not reveal this filename to the client. **[A/D/T]** Define a "whitelist" of allowed file types and reject any attempts to upload files of other types. **[D/T]** Review the configuration of the web server before deploying a web application and disable any unnecessary interpreters.	• Implement Sandboxing • Threat Modeling	CWE-676

No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
36	As a(n) architect/developer, I want to ensure **AND** as QA, I want to verify that the system does not allow use of potentially dangerous functions	**[D/T]** Periodically test the application with a static analysis tool designed to detect dangerous or risky functions for the particular language in which the application is written. Replace any detected dangerous functions with their safe equivalents. Some examples of dangerous functions include (but are not limited to): • C/C++: § strcpy § memcpy • PHP: § eval § exec • JavaScript: § eval **[T]** Periodically test the application interfaces (including file and network parsing code) with a fuzz testing tool; investigate any crashes the fuzzer generates. **[A/D/T]** If they are available, compile and link your application with any automatic memory protection options such as address space layout randomization (ASLR) and NX (No eXecute)-bit support.	• Minimize Use of Unsafe String and Buffer Functions • Use a Current Compiler Toolset • Use Static Analysis Tools • Perform Fuzz/Robustness Testing	CWE-676

Section 2b) Operational Security Tasks

Operational tasks are not directly tied to security stories, but are handled like continuous maintenance work or something requiring special attention in a sprint. Sometimes these are not tracked in the backlog but are part of the normal way of working within the team. It may make sense to use the backlog if the task ties up sprint team resources.

As an example, the task "Resolution of critical and high-severity issues identified by static code analysis tools" might turn into "Fix and verify the critical flaw identified by the static code analysis tool" that would go into the backlog and eventually the Scrum Team. The original generic task would still remain in the backlog.

As with **Section 2a**, these tasks are based on the observations the authors (or their internal peers working in the software security domain) have made while working with different software development teams within their respective organizations. Representatives from across SAFECode's membership also reviewed these tasks to ensure their practicality and broad applicability.

No.	Operational Security Task	Requirement/Recommendation
1	Configure bug tracking to track security vulnerabilities	Requirement for software development team
2	Verify security POCs and plan for fixes	Recommendation for software development team
3	Use latest compiler versions	Recommendation for new code
4	Resolve critical and high severity issues identified by static code analysis tools	Requirement for new code and recommendation for existing code
5	Keep track of patches/fixes to third party dependencies	Requirement for new as well as existing code
6	Keep track of patches/fixes to OS components	Requirement for new as well as existing code
7	Perform stricter code review of 'risky' code **	Requirement for new as well as existing code
8	Use appropriate security-related flags for compiler	Requirement for new as well as existing code
9	Continuously verify coverage of static code analysis tools	Requirement for new as well as existing code
10	Perform (and add to testing cycle) automated vulnerability scanner (OS and web as appropriate)	Requirement for software development team
11	Perform (and add to release cycle) automated malware scanner on released binaries	Requirement for software development team
12	Use secure versions of communication protocols	Requirement for new code and recommendation for existing code

No.	Operational Security Task	Requirement/Recommendation
13	Ensure inclusion of security patches/fixes applied in previous release(s)	Requirement for software development team
14	Ensure all developers have obtained secure coding training	Requirement for software development team
15	Ensure all QA engineers have obtained secure testing training	Requirement for software development team
16	Ensure security fixes are verified by security experts before committing them	Requirement for software development team
17	Periodically check to ensure that your SSL certificates (and all certificates above it in its chain of trust) have not expired or been revoked	Requirement for operational team

** Type of code that fits this category is as follows:

No.	Risky code category requiring a focused code review
1	Windows services and *nix daemons listening on network connections
2	Windows services/applications running as SYSTEM/Admin or *nix daemons running as root
3	Code listening on unauthenticated network ports connections
4	ActiveX controls
5	User (direct or indirect) input validation code
6	setuid root applications on *nix
7	Code that parses data from non-admin/non-root writeable files, email attachments, /temp directory, web downloaded files, log viewer code, event viewer code, report generators, file paths values, files that use UNC, lock files, application critical files such as config. files, dll load paths
8	Code that interfaces with third-party module(s)
9	Code that interfaces with C/C++/any ordinary executable files
10	Code dealing with authentication/authorization/encryption/any other core security features
11	Code that implements proprietary protocol OR handles proprietary implementation of standard protocols

Section 3) Tasks Requiring the Help of Security Experts

This section outlines advanced security tasks that typically require guidance from software security experts (in-house or consultants) for the first few iterations or in an ongoing manner.

The tasks here behave somewhat similarly to the ones in **Section 2a** and **2b**, except for the ones marked as "first few iterations" which would be closed and thus removed from the backlog once the team is comfortable performing that task as needed. These types of tasks relate more to the competencies of the team members and their way of working than actual tasks, and might only be needed once or twice in the backlog.

The value of tracking them in the backlog is often time-management. For example, consider the security task "Building a basic set of security test cases with the assistance of a security specialist." The Product Owner would need to have visibility into this as it would have a direct impact on the throughput of the Scrum Team.

No.	Security Experts to Help	Frequency of Help
1	Software security training (secure coding and secure testing)	Always
2	Security fix/patch validation (completeness and strength)	Always
3	Performing threat modeling for new/enhanced features	First few iterations
4	Conduct penetration tests on the software around beta stage	First few iterations
5	Enhance existing test suite to include security test cases	First few iterations
6	Perform file fuzz testing	First few iterations
7	Perform network fuzz testing	First few iterations
8	Third-party security assurance (module/library)	First few iterations
9	Security tool recommendations and effective use (including customization)	First few iterations
10	Prioritization of resolution of issues identified by code analysis tools (static and run-time), especially for existing code	First few iterations
11	Environment hardening (development systems, building environment, deployment)	First few iterations
12	Securing configuration, e.g., web server hardening, ACLs on folders holding sensitive data, configuration file hardening, etc.	First few iterations

Appendix A) Residual Risk Acceptance

Residual risks are risks that remain after performing risk management (risk identification, risk/threat analysis, risk mitigation). They may comprise secondary risks, as well as accepted risks that have a mitigation plan, contingency plan, or fallback plan. Residual risk acceptance is the acceptance, after scrutiny, of exposure to residual risks.

Assuming a Scrum framework, there can be at least two distinctive residual risk acceptance gateways; the Scrum Team, and, assuming the role exists in an organization, the Product Owner. Depending on the organization, there could be a hierarchically-higher third residual risk acceptance gateway, a Business Owner who is ultimately responsible for approving residual risks from the overall business perspective. Usually, the Business Owner is the Customer, with the Product Owner being the Customer's proxy.

To be able to decide whether a sprint is done or not done, the Scrum Team must add security to the criteria that determine completion of a sprint. This criteria list is known as the "Definition of Done" and usually consists of other non-functional needs as well; in effect it's a sprint-specific quality gate.

During each Sprint Review (whereby the Scrum Team presents the result of a sprint to the Product Owner), the team must inquire: *"Given the security-focused stories **(Section 2a)** and operational security tasks **(Section 2b)** we picked up, can we say we're done from security perspective?"* and subsequently declare that the security tasks for that Sprint Review are either "done" or "not done." If a risk is identified, there are two alternatives: 1) a control that would mitigate the risk is added to the backlog; for example, a security feature, a requirement for creating some test cases, and/or an architectural change; or, 2) it is accepted as residual risk. Ignoring a risk corresponds to accepting a risk.

The capability of the Scrum Team to reason with the Product Owner that a sprint is "done" or "not done" from the security (and hence quality) perspective is contingent upon empowerment from the team's skill competencies, maturity and its bestowed authority and trust. Essentially this is the lowest level gateway of residual risk acceptance **(level 1)**.

The Product Owner has the critical role of being the next **(level 2)** hierarchical residual risk acceptance gateway above the Scrum Team. He/she needs to approve/disapprove the residual risk based on the following:

1. Whether all of the security features identified in the requirements management phase have been "done," and

2. Whether the implementation teams have raised any residual risks that they haven't been able to address.

The raised risks ought to be visible in the Software Backlog as controls that haven't yet been implemented.

Since the Product Owner has visibility into the entire software embodiment, he/she also needs to evaluate the Scrum Team's reasoning that the sprint is "done" from the security perspective, and compare that evaluation with their own past decisions in terms of the security requirements identified from the epic-derived features.

Additionally, the Customer for the software should be involved in the software-level residual risk acceptance decision.

If any remaining residual risk cannot be accepted by the Product Owner and/or the Customer, the code cannot be released and therefore the Product Owner must add new tasks to the backlog to take care of those risks.

Further, if there is a hierarchically higher **(level 3)** residual risk acceptance gateway, i.e., a Business Owner who is ultimately responsible for approving residual risks from the overall business perspective, the Business Owner would need to make that decision based on the Product Owner's and Customer's software-level residual risk acceptance decision with supporting substantiation.

Glossary

Business Owner: Usually the customer who dictates the overall business objective of the system/application being developed.

Critical Resource: Any kind of system object and/or actor whose content or existence may impact the correct function of the system, and/or the confidentiality and integrity of its data. For example, an IPC semaphore used for synchronizing resources, or a file containing password hashes.

CWE-ID: Common Weakness Enumerations
http://cwe.mitre.org/
Created by MITRE Corp., CWE provides a unified, measurable set of software weaknesses that can help enable more effective discussion, description, selection and use of software security practices.

PCI DSS: Payment Card Industry Data Security Standard
https://www.pcisecuritystandards.org/security_standards/
This standard was created to increase controls around cardholder data to reduce credit card fraud via its exposure. Compliance is a must for organizations that handle cardholder information in any form.

OWASP: Open Web Application Security Project is a free, open-to-all community with local chapters worldwide aiming to improve the security of web applications.

POC: Proof-of-Concept is a realization of a certain method or idea to demonstrate its feasibility. Security POC is a working exploit of a vulnerability observed in a system/application.

Product Owner: Usually someone from marketing, product management, or anyone with a solid understanding of users, the market place, the competition, and of future trends for the domain or type of system/application being developed. This person is responsible for prioritizing the software backlog.

References

- SAFECode's Fundamental Practices for Secure Software Development
- 2011 CWE/SANS Top 25 Most Dangerous Development Errors
- 2011 CWE/SANS 16 Weakness On the Cusp
- OWASP Top 10
- Agile Development using Microsoft's Security Development Lifecycle
- Security in Agile PLC – Practical Navigational Aid for Speed Boats
- Software Security in Agile Product Management

PRIMARY AUTHORS

Vishal Asthana, Symantec Corporation
vishal_asthana@symantec.com

Izar Tarandach, EMC Corporation
izar.tarandach@rsa.com

Niall O'Donoghue, Nokia Corporation
niall.odonoghue@nokia.com

Bryan Sullivan, Microsoft Corporation
bryans@microsoft.com

Mikko Saario, Nokia Corporation
mikko.saario@nokia.com

CONTRIBUTORS

Reeny Sondhi, EMC Corporation
reeny.sondhi@emc.com

Edward Bonver, Symantec Corporation
edward_bonver@symantec.com

Matthew Coles, EMC Corporation
matthew.coles@emc.com

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 members include Adobe Systems Incorporated, EMC Corporation, Juniper Networks, Inc., Microsoft Corporation, Nokia, SAP AG, Siemens AG and Symantec Corp.

For more information, please visit www.safecode.org.

SAFECode
Software Assurance Forum for Excellence in Code
(p) +1 781-876-8833 (f) +1 781-224-1239
feedback@safecode.org
www.safecode.org
Twitter: @SAFECodeForum
Facebook: www.facebook.com/SAFECode