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

Fundamental Practices for Secure Software Development

A Guide to the Most Effective Secure Development Practices in Use Today

OCTOBER 8, 2008

Editor Stacy Simpson, SAFECode

Contributors

Gunter Bitz, SAP AG
Jerry Cochran, Microsoft Corp.
Matt Coles, EMC Corporation
Danny Dhillon, EMC Corporation
Chris Fagan, Microsoft Corp.
Cassio Goldschmidt, Symantec Corp.
Wesley Higaki, Symantec Corp.
Michael Howard, Microsoft Corp.

Steve Lipner, Microsoft Corp.
Brad Minnis, Juniper Networks, Inc.
Hardik Parekh, EMC Corporation
Dan Reddy, EMC Corporation
Alexandr Seleznyov, Nokia
Reeny Sondhi, EMC Corporation
Janne Uusilehto, Nokia
Antti Vähä-Sipilä, Nokia

Executive Summary

Software assurance encompasses the development and implementation of methods and processes for ensuring that software functions as intended while mitigating the risks of vulnerabilities and malicious code that could bring harm to the end user. Recognizing that software assurance is a vital defense in today's increasingly dynamic and complex threat environment, leading software vendors have undertaken significant efforts to reduce vulnerabilities, improve resistance to attack and protect the integrity of the products they sell. These efforts have resulted in significant improvements in software security and thus offer important insight into how to improve the current state of software security.

Through its analysis of the individual software assurance efforts of its members, SAFECode has identified a core set of secure development practices that can be applied across diverse development environments to improve software security. It is important to note that these are the "*practiced practices*" employed by SAFECode members. By bringing these methods together and sharing them with the larger community, SAFECode hopes to move the industry beyond defining sets of often-cited, but rarely-used, best practices to describing sets of software engineering disciplines that have been shown to improve the security of software and are currently in common practice at leading software companies. Using this approach enables SAFECode to encourage the adoption of best practices that are proven to be both effective and implementable even when different product requirements and development methodologies are taken into account.

A key goal of this paper is to keep it short, pragmatic and highly actionable. It prescribes specific security practices at each stage of the development process—Requirements, Design, Programming, Testing, Code Handling and Documentation—that can be implemented across diverse development environments.

Software vendors have both a responsibility and business incentive to ensure product assurance and security. SAFECode has collected, analyzed and released these security best practices in an effort to help others in the industry to initiate or improve their own software assurance programs and encourages the industry-wide adoption of the secure development methods outlined in this paper.

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

Table of Contents

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

Overview of Best Practices for Secure Software Development

There are several different software development methodologies in use today. However, they all share common elements from which we can build a nearly universal framework for software development.

A review of the security-related disciplines used by the highly diverse SAFECode members reveals that there are corresponding security practices for each stage of the software development lifecycle that can improve software security and integrity, and are applicable across diverse environments. The examination of these vendor practices reinforces the assertion that software assurance must be addressed throughout the software development lifecycle to be effective and not treated as a one-time event or single box on a check list. Moreover, all of these security practices are currently being used by SAFECode members, a testament to their ability to be integrated and adapted into real-world development environments even when unique product requirements are taken into account.

The practices defined in this document are as diverse as the SAFECode membership, spanning web-based applications, shrink-wrapped applications, database applications as well as operating systems and embedded systems.

To aid others within the software industry in adopting and using these software assurance best practices effectively, this paper describes each identified security practice across the software development lifecycle and offers implementation advice based on the experiences of SAFECode members.

> This paper describes each identified security practice across the software development lifecycle and offers implementation advice based on the experiences of SAFECode members.

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

Requirements

During requirements definition, a set of activities is defined to formalize the security requirements for a specific product release. These practices identify functional and non-functional requirements, and include conducting a product or code-specific risk assessment, identifying specific security requirements to address the identified risks, and defining the security development roll-out plan for that release. The product development team first identifies security requirements from use cases, customer inputs, company policy, best practices and security improvement goals. Then, the team prioritizes security requirements based on threat and risk levels such as threats to code integrity, intellectual property protection, personally-identifiable information (PII) or sensitive data, features that require admin/root privileges and external network interfaces.

The security engineering requirements help drive design, programming, testing, and code handling activities similar to those outlined in the rest of this document. It is also useful to review security requirements that were "deferred" from the previous release and prioritize them with any new requirements.

During requirements definition, it is important that the product managers and other business leaders who allocate resources and set schedules are aware of the need to account for time to engage in secure development practices.

Awareness training and "return on investment" arguments help present the business case for secure development. It is important that these decision-makers understand the risks that their customers will have to accept should too little effort be put into secure development.

In preparation for each product release, the development and QA staff members should be trained in secure development and testing. Training goals help track and drive improvement in this area.

The security requirements cover areas such as:

- Staffing requirements (background verification, qualifications, training and education, etc.)
- Policy on disclosure of information and project confidentiality
- Authentication and password management
- Authorization and role management
- Audit logging and analysis
- Network and data security
- Third party component analysis
- Code integrity and validation testing
- Cryptography and key management
- Data validation and sanitization
- Serviceability
- Ongoing education and awareness

Design

The single secure software design practice used across SAFECode members is threat analysis, which is sometimes referred to as "threat modeling" or "risk analysis." Regardless of the name, the process of understanding threats helps elevate potential design issues that are usually not found using other techniques such as code reviews and static source analyzers. In essence, threat analysis helps find issues before code is committed so they can be mitigated as early as possible in the software development lifecycle. For example, rather than wait for an analysis tool to potentially find injection vulnerabilities, it's better for a development team to realize that their product may be vulnerable to these issues and put in place defenses and coding standards to reduce the risk from the start.

If an organization does not have expertise in building threat models, a free-form discussion is better than not thinking at all about potential application weaknesses. Such "brainstorming" should not be considered a complete solution, and should only serve as a stepping stone to a more robust threat analysis method.

The risk of not doing an adequate job of identifying architectural and design security flaws is that customers, researchers or attackers may find these flaws which would then require a major upgrade or re-architecture effort to mitigate the resulting vulnerability—an extremely costly venture.

Some SAFECode members have adopted "misuse cases" to help drive their understanding of how attackers might attack a system.

To get the full benefit of threat modeling while designing the software, software designers and architects should strive to mitigate any identified issues before moving beyond design whenever possible. Comprehensive treatment of mitigation techniques is beyond the scope of this paper, but most secure design practices today are based on the fundamental work by Saltzer and Schroeder.

SAFECode members also recommend selecting standard, proven security toolkits, such as cryptographic and protocol libraries, during the requirements or design phase and advise development groups to avoid building their own security technologies and protocols.

Resources

- *The Security Development Lifecycle.* Chapter 9, "Stage 4: Risk Analysis" Microsoft Press, Howard & Lipner.

- *The Protection of Information in Computer Systems.* Proceedings of the IEEE, 63(9):1278–1308, September 1975. J.H. Saltzer and M.D. Schroeder.

- *Software Security Assurance: State-of-the-Art Report.* Section 5.2.3.1, "Threat, Attack, and Vulnerability Modeling and Assessment" Information Assurance Technology Analysis Center (IATAC), Data and Analysis Center for Software (DACS).

- *Security Mechanisms for the Internet.* Bellovin, Schiller, Kaufman; ftp://ftp.rfc-editor.org/in-notes/rfc3631.txt

- *Capturing Security Requirements through Misuse Cases*, Sindre and Opdahl; http://folk.uio.no/nik/2001/21-sindre.pdf

Programming

Throughout programming, the following practices
are used across the majority of SAFECode members:

- Minimize unsafe function use

- Use the latest compiler toolset

- Use static and dynamic analysis tools

- Manual code review

- Validate input and output

- Use anti-cross site scripting libraries

- Use canonical data formats

- Avoid string concatenation for dynamic SQL

- Eliminate weak cryptography

- Use logging and tracing

These practices are detailed on the following pages.

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

Minimize unsafe function use

Buffer overrun vulnerabilities are a common and easy-to-introduce class of vulnerability that primarily affects C and C++. An analysis of buffer overrun vulnerabilities over the last ten years shows that a common cause is using unsafe string- and buffer-copying C runtime functions. Functions such as, but not limited to, the following function families are actively discouraged by SAFECode members in new C and C++ code, and should be removed over time from older code.

Function Families to Remove:

- strcpy family
- strncpy family
- strcat family
- strncat family
- scanf family
- sprint family
- gets family

Development engineers can be trained to avoid using these function calls, but using tools to search the code for these calls helps validate the training efforts and identify problems in legacy code. Building the execution of these tools into the "normal" compile/build cycles relieves the developers from having to take "special efforts" to meet these goals.

Finally, it is important to be aware of library or operating system specific versions of these functions. For example Windows has a functional equivalent to strcpy called lstrcpy and Linux has strcopy, to name but a few, and these too should be avoided.

Resources

- Security Development Lifecycle (SDL) Banned Function Calls;
  http://msdn.microsoft.com/en-us/library/bb288454.aspx
- strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation, Miller & de Raadt;
  http://www.usenix.org/events/usenix99/millert.html

When possible, use the latest compiler toolset to take advantage of compile-time and run-time defenses

As previously noted, a very common and dangerous type of vulnerability that primarily affects code written in C or C++ is the buffer overrun. It is easy to fix most buffer overrun vulnerabilities by moving to languages other than C and C++, but that is much harder to do in practice because for many classes of software, C and C++ are the perfect languages for the job. Because many vulnerabilities in C and C++ are serious, it is important to use C and C++ compilers that offer compile-time and run-time defenses against buffer overruns automatically. Examples include:

Microsoft Visual C++ 2005 SP1 and later offers:

- /GS for stack-based buffer overrun defenses
- /DYNAMICBASE for image and stack randomization
- /NXCOMPAT for CPU-level No-eXecute (NX) support
- /SAFESEH for exception handler protection
- Warning C4996 for insecure C runtime function detection and removal

gcc 4.1.2-25[1] and later offers:

- –fstack-protector for stack-based buffer overrun defenses
- –Wl, –pie for image randomization
- –D_FORTIFY_SOURCE=2 and –Wformat-security for insecure C runtime function detection and removal

Development teams can decide to use these compiler flags on every compile session or on selected sessions depending on their individual circumstances. It is important that any errors generated by these complies are analyzed and addressed.

Resources

- Protecting Your Code with Visual C++ Defenses;
  http://msdn2.microsoft.com/en-us/magazine/cc337897.aspx

- Object size checking to prevent (some) buffer overflows;
  http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

- Exploit Mitigation Techniques;
  http://www.openbsd.org/papers/ven05-deraadt/index.html

---

1 Not all versions of gcc on all platforms offer all defenses. Also, Apple has backported some defenses to gcc 4.01 on the Macintosh OS X platform.

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

Use static and dynamic code analysis tools to aid code review process to find vulnerabilities

Source code and binary analysis tools are now becoming commonplace, and the use of such tools is highly recommended to find common vulnerability types. These tools are adjunct to manual code review, not a replacement.

The state-of-the-art of these tools requires that developers analyze sometimes voluminous results that may contain many false positives. Considerable tuning may be required to get the most benefit from these tools. It also seems that tools from different vendors catch different types of issues; that is, no one tool today finds all faults. There is some up-front investment required to get the greatest benefit from these tools, but the effort is worthwhile.

Resources

- *The Security Development Lifecycle.* Chapter 21, "SDL-Required Tools and Compiler Options" Microsoft Press, Howard & Lipner.

- Detecting and Correcting C/C++ Code Defects;
  http://msdn.microsoft.com/en-us/library/ms182025.aspx

- Using Static Analysis for Software Defect Detection;
  http://video.google.com/videoplay?docid=-8150751070230264609

- FxCop; http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx

- List of tools for static code analysis;
  http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

- Static Analysis Tools; http://www.securityinnovation.com/pdf/si-report-static-analysis.pdf

Manually review code after security education

Manual code review, especially review of high-risk code, such as code that faces the Internet or parses data from the Internet, is critical, but only if the people performing the code review know what to look for and how to fix any code vulnerabilities they find. The best way to help understand classes of security bugs and remedies is education, which should minimally include the following areas:

- C and C++ vulnerabilities and remedies, most notably buffer overruns and integer arithmetic issues.
- Web-specific vulnerabilities and remedies, such as cross-site scripting (XSS).
- Database-specific vulnerabilities and remedies, such as SQL injection.
- Common cryptographic errors and remedies.

Many vulnerabilities are programming language (C, C++ etc) or domain-specific (web, database) and others can be categorized by vulnerability type, such as injection (XSS and SQL Injection) or cryptographic (poor random number generation and weak secret storage) so specific training in these areas is advised.

Resources

- A Process for Performing Security Code Reviews, Michael Howard, IEEE Security & Privacy July/August 2006.
- .NET Framework Security — Code Review; http://msdn.microsoft.com/en-us/library/aa302437.aspx
- *Common Weakness Enumeration*, MITRE; http://cwe.mitre.org/
- Security Code Reviews; http://www.codesecurely.org/Wiki/view.aspx/Security_Code_Reviews
- Security Code Review — Use Visual Studio Bookmarks To Capture Security Findings; http://blogs.msdn.com/alikl/archive/2008/01/24/security-code-review-use-visual-studio-bookmarks-to-capture-security-findings.aspx
- Security Code Review Guidelines, Adam Shostack; http://www.verber.com/mark/cs/security/code-review.html
- OSWASP Top Ten; http://www.owasp.org/index.php/OWASP_Top_Ten_Project

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

Validate input and output to mitigate common vulnerabilities

Simply checking the validity of incoming data and rejecting non-conformant data can remedy the most common vulnerabilities. In some cases checking data validity is not a trivial exercise, but is critically important to understanding the format of incoming data to make sure it is correct. For text- and XML-based data, software can use regular expressions or string comparisons for validation. Binary data is harder to verify, but at a minimum, code should verify data length and field validity.

In some applications types, notably web-based applications, validating and/or sanitizing output is important and can help mitigate classes of vulnerabilities such as cross-site scripting, HTTP response splitting and cross-site request forgery vulnerabilities.

Resources
- *Writing Secure Code 2nd Ed.* Chapter 10, "All Input is Evil!" Michael Howard & David LeBlanc, Microsoft Press.
- ASP.NET Input and Data Validation;
  http://wiki.asp.net/page.aspx/45/input-and-data-validation/

Use anti-cross site scripting (XSS) libraries

As a defense-in-depth measure, using anti-XSS libraries is very useful. In its simplest form, a minimal anti-XSS defense is to HTML encode all web-based output that may include untrusted input; however, more secure libraries also exist, such as those in the resources section below.

Resources
- OWASP PHP AntiXSS Library; http://www.owasp.org/index.php/Category:OWASP_PHP_AntiXSS_Library_Project
- Microsoft Anti-Cross Site Scripting Library V1.5: Protecting the Contoso Bookmark Page, Kevin Lam;
  http://msdn.microsoft.com/en-us/library/aa973813.aspx

Use canonical data formats

Where possible, applications that use resource names for filtering or security defenses should use canonical data forms. Canonicalization describes the mechanisms to derive a canonical expression from different polymorphic expressions. For example, within the context of a search engine, the data file 'Hello World.doc' may be accessible by any one of the following polymorphic links:

http://www.site.com/hello+world.doc
http://www.site.com/hello%20world.doc
http://www.site.com:80/hello%20world.doc

The canonical representation ensures that the various forms of an expression (for example, URL encoding or Unicode escapes) do not bypass any security or filter mechanisms. A polymorph representation of data is not necessarily an attack in itself, but may help to slip malicious data past a filter or defense by "disguising" it. There are many canonicalization vulnerabilities, including, path traversal and URL bypass.

Resources

- *Writing Secure Code 2nd Ed.* Chapter 11, "Canonical Representation Issues" Michael Howard & David LeBlanc, Microsoft Press.

- OWASP Canonicalization, Locale and Unicode;
  http://www.owasp.org/index.php/Canonicalization%2C_locale_and_Unicode

- How to programmatically test for canonicalization issues with ASP.NET;
  http://support.microsoft.com/kb/887459

Avoid string concatenation for dynamic SQL statements

Building SQL statements is common in database-driven applications. Unfortunately the most common way and the most dangerous way to build SQL statements is to concatenate untrusted data with string constants to build SQL statements. Except in very rare instances, string concatenation should not be used to build SQL statements. Rather, developers should use SQL placeholders or parameters to build SQL statements securely. Different programming languages, libraries and frameworks offer different functions to create SQL statements using placeholders or parameters. As a developer it is important to understand how to use this functionality correctly.

Resources

- Giving SQL Injection the Respect it Deserves, Michael Howard;
  http://blogs.msdn.com/sdl/archive/2008/05/15/giving-sql-injection-the-respect-it-deserves.aspx

Eliminate weak cryptography

Over the last few years, serious weaknesses have been found in many cryptographic algorithms. Weak security controls in general should be avoided, whether the weaknesses are in authentication, authorization, logging, encryption or data validation/sanitization.

Only proven algorithms and implementations should be used. US Federal government customers require FIPS 140-2 validation for products using cryptography. FIPS 140-2 defines a set of algorithms that have been determined to be sound. Vendors also need to consider cryptographic export restrictions, but FIPS 140-2 provides a sound standard to consider.

The following algorithms and cryptographic entities should be treated as insecure:

- Embedded private data, passwords, keys or key material.
- MD4
- MD5
- SHA1
- Symmetric keys less than 128-bits long (that means DES is too weak as it supports only a 56-bit key).
- Use of stream ciphers (such as ARC and RC4) is discouraged owing to subtle weaknesses in the way stream ciphers are often used.
- Any cryptographic algorithm you have invented yourself or has not been subject to academic peer review.
- Block ciphers using Electronic Code Book (ECB) mode.

Resources

- *The Security Development Lifecycle.* Chapter 20, "SDL Minimum Cryptographic Standards" Microsoft Press, Howard & Lipner.
- National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 140-2; http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
- Public-Key Cryptography Standards (PKCS); http://www.rsa.com/rsalabs/node.asp?id=2124
- Public-Key Infrastructure (X.509) (pkix); http://www.ietf.org/html.charters/pkix-charter.html

Use logging and tracing

Logging and tracing are important elements for securing, monitoring and debugging applications.

Administrators are the main users of the logging system and traces are used by developers and the support organization. Logging systems should record data that pertains to the normal operation of the system including successful and failed events. Tracing systems should record data that might help pinpoint a bug in the system.

It is critically important that logs and trace files do not contain sensitive data such as passwords.

Resources

- OWASP Reviewing Code for Logging Issues;
  http://www.owasp.org/index.php/Reviewing_Code_for_Logging_Issues
- OWASP Error Handling, Auditing and Logging;
  https://www.owasp.org/index.php/Error_Handling%2C_Auditing_and_Logging

Testing

Testing activities validate the secure implementation of a product, which reduces the likelihood of security bugs being released and discovered by customers and/or malicious users. The majority of SAFECode members have adopted the following software security testing practices in their software development lifecycle. The goal is not to "test in security," but rather to validate the robustness and security of the software products prior to making the product available to customers. These testing methods do find security bugs, especially for products that may not have undergone critical secure development process changes.

Fuzz testing

Fuzz testing is a reliability and security testing technique that relies on building intentionally malformed data and then having the software under test consume the malformed data to see how it responds. The science of fuzz testing is somewhat new but it is maturing rapidly. There is a small market for fuzz testing tools today, but in many cases software developers must build bespoke fuzz testers to suit specialized file and network data formats. Fuzz testing is an effective testing technique because it uncovers weaknesses in data handling code.

Resources

- Fuzz Testing of Application Reliability, University of Wisconsin;
  http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html

- Automated Whitebox Fuzz Testing, Michael Levin, Patrice Godefroid and Dave Molnar, Microsoft Research;
  ftp://ftp.research.microsoft.com/pub/tr/TR-2007-58.pdf

- IANewsletter Spring 2007 "Look out! It's the fuzz!" Matt Warnock;
  http://iac.dtic.mil/iatac/download/Vol10_No1.pdf

- *Fuzzing: Brute Force Vulnerability Discovery.* Sutton, Greene & Amini, Addison-Wesley.

- *Open Source Security Testing Methodology Manual.* ISECOM.

- *Common Attack Pattern Enumeration and Classification,* MITRE;
  http://capec.mitre.org/

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

Penetration testing and third-party assessment

The goal of penetration testing is to find security issues in an application by applying testing techniques usually employed by attackers. Some SAFECode members have dedicated penetration testing teams while others employ external penetration and security assessment companies. Some SAFECode members use both in-house and external security penetration expertise. Internal QA teams should perform security testing along with standard functional testing as part of a comprehensive test plan.

While there is significant value in having an objective analysis of the security of the system, it is important to realize that a penetration test cannot make up for an insecure design or poor development and testing practices.

The advantage of using competent, third-party penetration testers is their breadth of experience. The challenge is finding third-party testers that will do a complete job for your specific product type, architecture or technologies. Developing an in-house penetration team has the advantage of maintaining internal product knowledge from one test to the next. However, it takes time for an internal team to develop the experience and skill sets to do a complete penetration testing job and penetration testing should be prioritized after secure design and coding and other security testing measures.

Use of automated testing tools

Automation at all stages of the development process is important because automation can tirelessly augment human work. During testing, the most common tools used by SAFECode members include:

- Fuzzing tools
- Network vulnerability scanners
- Web application vulnerability scanners
- Packet analyzers
- Automated penetration testing tools
- Network/web proxies that manipulate network data
- Protocol analysis
- Anti-malware detection on final media

The first three help development teams look for code vulnerabilities, and the last helps to verify that the final executable images are free from known malicious code.

Code Integrity and Handling

Software integrity and code handling practices increase confidence in software products and applications by reducing the risk of malicious code being present. The principles outlined below exist in the context of other IT functions such as backup and recovery, business continuity services, physical and network security and configuration management systems.

These practices derive from established integrity principles:

- Least Privilege Access
- Separation of Duties
- Chain of Custody and Supply Chain Integrity
- Persistent Protection
- Compliance Management

In this context software integrity practices address access, storage and handling during software development processes which include procurement, code and test, build, release and distribution. Controls must be in place to assure the confidentiality, integrity and availability of code throughout its lifecycle (including across the supply chain).

- Source code should be kept in well protected source code control systems (Repositories, Build Systems, Software Configuration Management) with strong authentication and role-based access control following the principle of "least privilege."

- The chain of custody of code throughout its lifecycle should be verifiable to establish the origin of each change made during the source code's lifetime.

- Code should be protected while active, at rest and in transit to obstruct attempts to tamper with code, and when they occur changes are evident and reversible.

- Event and audit logs generated by applications and network devices should be closely monitored and analyzed. Done correctly and consistently, log analysis is a reliable and accurate way to discover potential threats and identify malicious activity. In addition to malicious attacks, event and audit log management can highlight administrative actions performed by well-meaning IT staff that have unintended consequences.

- Code should be verifiable for its integrity and authenticity by consumers (e.g. signed code).

- Bugs in code that create vulnerabilities must be resolved promptly and continuously throughout code's lifecycle (including throughout its sustainment phase).

Documentation

Before deploying software, administrators must understand the security posture of the software; this might include knowing which ports to allow through a firewall, or operating system changes to make the software work correctly.

An issue that many customers have requested is more information on how to securely configure their software either "out of the box" or using wizards or more documentation for given environments.

Documentation defining the software security best practices is the prime source of information for administrators. The documentation can be as simple as a set of "Do's and Don'ts" or as complete as a large book defining every possible security setting and the security and usability implications of those settings.

Conclusion

Improving software security requires software development process improvements along the entire software development timeline, not just random one-time events or simple code review. SAFECode members recognize this and have adopted a core set of improvements that demonstrably improve security. It is recommended that all software vendors, irrespective of target operating system, customer type or development environment adopt the practices laid out in this document.

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 Corp., Nokia, SAP AG and Symantec Corp. For more information, please visit www.safecode.org.

SAFECode	(p) 703.812.9199
2101 Wilson Boulevard	(f) 703.812.9350
Suite 1000	(email) stacy@safecode.org
Arlington, VA 22201	www.safecode.org

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