Secure Coding

Software Assurance Pocket Guide Series:
Development, Volume VI
Version 1.1, February 22, 2011

Software Assurance (SwA) Pocket Guide Resources

This Guide provides a ground-level resource for creating code that is secure and operates as expected. As part of the Software Assurance (SwA) Pocket Guide series, this summary Guide is offered for information only. For details, see referenced source documents. For proper attribution, please include mention of these sources when referencing any part of this document.

This volume of the SwA Pocket Guide series focuses on secure coding principles and practices that mitigate vulnerabilities and support overall software security. It describes basic concepts and principles for writing secure code. It addresses preparations for writing secure code, secure coding principles, secure coding practices, secure memory and cache management, secure error and exception handling, and what to avoid. It provides questions for managers in development and for procurement organizations to assess coding. The answers can help them establish whether the teams responsible for the delivery of software use the requisite practices: practices that contribute to the overall security of software.

The back of this pocket guide contains limitation statements, and a listing of additional topics covered in the SwA Pocket Guide series. All SwA Pocket Guides and SwA-related documents are freely available for download via the SwA Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa.

Acknowledgements

The SwA community collaborates to develop SwA Pocket Guides. The SwA Forum and Working Groups function as a stakeholder meta-community that welcomes additional participation in advancing and refining software security. All SwA-related information resources are offered free for public use. The SwA community invites your input: please contact Software.Assurance@dhs.gov for comments and inquiries. For the most current pocket guides, refer to the SwA community website at https://buildsecurityin.us-cert.gov/swa/.

Members from government, industry, and academia comprise the SwA Forum and Working Groups. The Groups focus on incorporating SwA considerations into acquisition and development processes to manage potential risk exposure from software and from the supply chain. Participants in the SwA Forum's Processes & Practices Working Group contributed to developing the material used in this pocket guide in an effort to encourage application of SwA practices throughout the Software Development Lifecycle (SDLC).

Information contained in this pocket guide is primarily derived from ***"Enhancing the Development Life Cycle to Produce Secure Software."*** Information for that and other references is listed in the *Resources* box that follows.

The Processes & Practices Working Group would like to thank the Department of Homeland Security (DHS), National Cyber Security Division's Software Assurance team for substantial support in completing this guide and related SwA documents. Additional thanks for providing material and input goes to Jason Grembi from Sterling Connect LLC, Robert Secord of the Carnegie Mellon University, Software Engineering Institute, David Wheeler from the Institute for Defense Analysis, and Art Conklin from the University of Houston and the Workforce Education and Training Working Group.

Resources

» BuildSecurityIn Coding Practices resources. 25 June 2010. United States Government. Department of Homeland Security. 28 July 2010 <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding.html>.

» BuildSecurityIn Coding Rules resources. 16 May 2008. United States Government. Department of Homeland Security. 28 July 2010 <https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html>.

» CERT Secure Coding Standards. 28 August 2010. Carnegie Mellon University Software Engineering Institute/Computer Emergency Response Team [CERT]. 31 August 2010 <https://www.securecoding.cert.org/>.

» "Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today." SAFECode.org. Ed. Stacy Simpson. 8 October 2008. The Software Assurance Forum for Excellence in Code [SAFECode]. 13 July 2010 <http://www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf>.

» Goertzel, Karen Mercedes; et al. "Software Security Assurance: State-of-the-Art Report (SOAR)." Information Assurance Technology Analysis Center [IATAC]. 31July 2007. 3 August 2010 <http://iac.dtic.mil/iatac/reports.jsp#SOAR>.

» "Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses V 1.3." Community Resources and Information Resources. 24 May 2009 Version 1.3. United States Government. Department of Homeland Security. 13 July 2010 <https://buildsecurityin.us-cert.gov/swa/pocket_guide_series.html#development>.

» Microsoft. "Writing Secure Code." Microsoft Security Developer Center. 28 July 2010 <http://msdn2.microsoft.com/en-us/security/aa570401.aspx>.

» The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software, Michael Howard and Steve Lipner, Microsoft Press (May 31, 2006)

» "OWASP Comprehensive, Lightweight Application Security Process (CLASP) project." OWASP.org. 3 July 2009. The Open Web Application Security Project [OWASP]. 19 July 2010 <http://www.owasp.org/index.php/Category:OWASP_CLASP_Project>.

» "OWASP Secure Coding Practices Quick Reference Guide." OWASP.org. November 2010 Version 2.0. . The Open Web Application Security Project [OWASP]. 17 November 2010 <http://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide>

» SANS Software Security Institute. SANS. 20 September 2010 <http://www.sans-ssi.org/>.

» CWE/SANS TOP 25 Most Dangerous Software Errors. SANS. V2.0 February 16, 2010, <http://www.sans.org/top25-software-errors/>.

» Seacord, Robert C. Secure Coding in C and C++. Upper Saddle River, NJ: Addison-Wesley. September 2005.

» "Secure Coding Guide." Mac OS X Reference Library. 12 February 2010. 28 July 2010 <http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf>.

Resources

- » United States Government. Department of Homeland Security - Software Assurance Forum Process and Practices Working Group. "Enhancing the Development Life Cycle to Produce Secure Software." The Data & Analysis Center for Software [DACS]. October 2008 Version 2.0. United States Government. Defense Technical Information Center. 24 June 2010 <http://www.thedacs.com/techs/enhanced_life_cycles/>.
- » Wheeler, David A. Secure Programming for Linux and Unix HOWTO -- Creating Secure Software. 3 March 2010 Version 3. 20 Dec 2010 <http://www.dwheeler.com/secure-programs/>.

Overview

Code that is not properly protected can allow an intruder to view sensitive information, change or delete valuable or important data, deny service, run programs, or plant malicious code within a software system. Because of the risks associated with software and the supply chain, it is critically important that precautions be taken through every process in the software development lifecycle.

Secure coding is a prerequisite for producing robustly secure software. The development of secure software is a complex endeavor and requires a systematic process. The most commonly exploited vulnerabilities are seemingly easily avoided defects in software. Producing secure code is not an absolute science because it depends on a wide range of variables, some of which cannot be easily or accurately measured. Such variables range from the language or platform being used to the nature of the software being developed or the data with which the software is meant to work. This guide does not prescribe answers for all possible situations. Rather, it discusses fundamental practices for secure coding, and lists resources that provide more information about these practices. Using these resources, practitioners can write more secure code for their particular environment. Secure coding fundamentals and tips include:

Preparing to Write Secure Code

Before writing code, the development team must determine how they plan to accomplish their assigned task. It is also important to remember that writing secure code needs the support of secure requirements, architecture, and design. To find more on these topics, reference "Requirements and Analysis for Secure Software" and "Architecture and Design Considerations for Secure Software." Both of these documents are part of the SwA Pocket Guide Series.

Choose a Language with Security in Mind

The choice of programming language is an important factor in software security. In many cases, legacy code, the available skill sets, or customer requirements, including the environment, may restrict programming language options. When evaluating and selecting the programming language(s), you should ask these key security-relevant questions:

- Which languages have your programmers used? Programmers are likely to make more errors when developing code in languages they have little experience using.
- Does the language help or inhibit the development of secure code?
- What secure coding standards are available for the language (if any)?
- What security issues exist in the language? What mitigation strategies exist? How effective are they?

Weighing language choices: In many situations, the choice of language can directly affect the system's security. The most prominent example is the effect of array bounds checking in Java versus C. Although most modern C compilers support runtime bounds checking, this feature (or lack thereof) has left a legacy of buffer-overflow-based vulnerabilities in web servers, operating systems, and applications. Java is without these specific vulnerabilities, but may not be appropriate for environments where performance and a small footprint are important (such as embedded devices and smart cards). A Java Virtual Machine (JVM) may introduce too much strain on the system, resulting in potential denials of service. It is security issues such as these that make it important to have an understanding of which weaknesses exist or are more likely to appear based on the choice of language.

"Safe" languages: Most "safe" languages (or language variants) are intended to help avoid problems with buffers, pointers, and memory management. The bounds-checking interfaces developed by Microsoft and included in the emerging C1X standard can mitigate vulnerabilities resulting from improper string management in existing C programs, If the program is expected to operate in a particularly exposed environment, on cell phones, or on the web, use a language that includes its own security model and self-protecting features (*e.g.*, Java, Scheme, Categorical Abstract Machine Language). Alternatively, implement a virtual machine on the host system to contain and isolate the program.

Advantages of static typing: Languages with static typing such as Java, Scheme, MetaLanguage (ML), F#, and Ada ensure that operations are only applied to values of the appropriate type. Type systems that support type abstraction let programmers specify new, abstract types and signatures for operations. Such typing prevents unauthorized code from applying operations inappropriate for the given values. In this respect, type systems, like software-based reference monitors, go beyond operating systems in that they can be used to enforce a wider class of system-specific access policies. Static type systems also enable offline enforcement through static type checking rather than checking based on each specific instance of operation. This allows the type checker to enforce rules that are difficult to enforce with online techniques.

Built-in security mechanisms: Newer languages, such as C# have a variety of security mechanisms built into the language including type safe elements, code access security and role-based security included via the .Net framework. Although the .Net framework and C# contain many elements that can assist in secure development, it is still incumbent on the development team to properly use these elements. It is important to note that there may be certain features of the language that will provide more security; however, other aspects may open new security vulnerabilities. Therefore, it is necessary to keep informed of any vulnerabilities or issues that may arise from use of the selected language.

Some languages can reduce the damage that un-trusted programs can do by limiting the system calls that the program can invoke. Perl does this through its "taint mode," which only allows user-supplied input that is explicitly specified as "untainted" by the programmer [Birznieks]. Java provides the JVM as a "sandbox" and does not allow an un-trusted program to act outside the JVM. For example, un-trusted Java applets cannot create a new process or read/write to the local disk [Venners].

Don't rely on language choice alone: While picking a secure programming language is important, many developers make the mistake of assuming that a secure language is a panacea for security vulnerabilities. Without a secure development process and developers who know how to code securely, however, software written in one language is as easily broken as it would be in any other [Rook].

Resources

» Birznieks, Gunther. CGI/Perl Taint Mode FAQ. 3 June, 1998. 20 July 2010 <http://gunther.web66.com/FAQS/taintmode.html>.

» Rook, David. "Your choice of programming language doesn't matter, they are all insecure!" Security Ninja: Security Research, News & Guidance. Realex Payments. 15 October 2010. <http://www.securityninja.co.uk/your-choice-of-programming-language-doesn't-matter-they-are-all-insecure>.

» United States Government. Department of Homeland Security - Software Assurance Forum Process and Practices Working Group. "Enhancing the Development Life Cycle to Produce Secure Software." The Data & Analysis Center for Software [DACS]. October 2008 Version 2.0. United States Government. Defense Technical Information Center. 24 June 2010 <http://www.thedacs.com/techs/enhanced_life_cycles/>.

» Venners, Bill. "Java's security architecture." JAVAWORLD. 1 August 1997. 20 July 2010 <http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html>.

Create a Secure Development Process

Secure software development is a process that involves many elements from requirements through design, coding and testing. A key element in the process is the way information passes between the different teams during software construction. This process must support secure coding efforts. [Howard 2006]
The secure development lifecycle defines process elements for the entire development effort from the very beginning to the end. With respect to the coding process, the typical elements defined include:

- Defined coding standards
- Compiler settings for secure coding
- Source code analysis tools and usage
- Approved and banned functions
- Secure coding checklist
- Security gate checklists

The primary purpose of the secure development lifecycle process is to engage the project management effort in the coordinated effort between all the parties to ensure software that meets the complete set of requirements—including security requirements. Expecting the development team to produce secure code without the coordinated efforts of all the other parties involved in the effort is unrealistic. Many security issues result not from syntactic errors, but from design issues. Proper coordination between the teams is the primary function of the secure development lifecycle process. A prime example of this cross-team communication comes in the form of Threat Modeling. The threat modeling technique analyzes potential threats to a system and uses the results to design mitigations into the product. This requires communication between several teams, but results software that embodies security thinking, thus thwarting potential threats. Additional information regarding coding standards and guidelines is discussed as part of the "Follow Secure Coding Standards and/or Guidelines" section.

For more information regarding secure software design, please reference "Architecture and Design Considerations for Secure Software," which is available in the SwA Pocket Guide Series library.

Create an Application Guide

Before the development team starts the coding process, the organization must establish policy in an application guide that is clear, understandable, and comprehensive enough to help other programmers write the specified code. This way any developer can easily maintain the code. The application guide should contain the following information [Grembi, 82]:

- Where to get specific software (i.e. URL address, servers)
- Where to install software (directory structure, hard drives, servers)
- Where to find license information
- Where to find deployment instructions
- How the software is tested and what tools to use
- Where to push code into production (server names, IP addresses)
- Procedures for checking in and checking out code
- Identification of language coding standards

Consistent code is fundamental; however it is not sufficient for achieving secure code. By following the guide, the developers can ease readability and increase the likelihood of working well together. If created correctly, this guide can be used throughout the organization. While the process may need to adjust to the needs of different projects or departments, this guide should help disseminate and enforce code development guidelines throughout the organization.

Resources

- Grembi, Jason. Secure Software Development: A Security Programmer's Guide. Boston: Course Technology, Cengage Learning. 2008.
- The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software, Michael Howard and Steve Lipner, Microsoft Press (May 31, 2006)

Identify Safe and Secure Software Libraries

A variety of safe/secure libraries have been developed for various languages and purposes. Many of these libraries have been developed for the C and C++ programming languages to provide replacements for standard library functions that are prone to misuse.

Types of Secure Libraries: Significant attention should be given to the sourcing of these libraries. Since the team often does not have the time to look over the code in the library to see if it is genuine and has not been tampered with, the files need to come from a trusted source. The definition of a trusted source depends on the team and the software being developed. Examples of libraries to consider are:

- **Computer Emergency Response Team's (CERT's)** managed string library [Burch 2010] was developed in response to the need for a string library that can improve the quality and security of newly developed C-language programs while eliminating obstacles to widespread adoption and possible standardization. As the name implies, the managed string library is based on a dynamic approach; memory is allocated and reallocated as required. This approach eliminates the possibility of unbounded copies, null-termination errors, and truncation by ensuring that there is always adequate space available for the resulting string (including the terminating null character).
- **SafeInt** is a C++ template class written by David LeBlanc [LeBlanc 04]. SafeInt tests the values of operands before performing an operation to determine whether errors might occur. The class is declared as a template, so it can be used with any integer type. SafeInt is currently used extensively throughout Microsoft, with substantial adoption within Office and Windows. If you are compiling for the Microsoft compiler only, a very similar version is now available with Visual Studio 2010. It can be used with any compiler that has good template support, and is known to work on Visual Studio 7.1 or later.
- **Open Web Applications Security Project's (OWASP) Enterprise Security API (ESAPI)** simplifies many security tasks such as input validation or access controls [Melton].
- **antiXSS** libraries are available for web applications. AntiXSS libraries focus on preventing Cross Site Scripting (XSS) attacks. These antiXSS libraries, at a minimum, allow the HTML-encoding of all output derived from un-trusted input. Examples include the OWASP PHP AntiXSS Library and the Microsoft Anti Cross Site Scripting Library [SAFECode 11].

Securing the libraries: After picking a secure library to work with, it is important to make sure that it stays secure. In the case of Java JAR files, it is easy to un-JAR the file, tamper with the class, and then re-JAR the file. When a problem is found, it could take weeks or months to determine that the problem is in the JAR file. For this reason libraries should be placed in a secure directory, with limited access rights, to make sure they are not tampered with [Grembi]. The goal is not to make the library impossible to repack, but to limit who can do it.

Code signing is a technology that can be employed to ensure that libraries and functions are not tampered with or changed. Methods for ensuring code integrity before use include Authenticode, strong naming, and Windows Side by Side (WinSxS). Code signing is also available in open-source code library tools, such as Mercurial and GIT. The key takeaway is that the development team needs to define and follow an appropriate code-signing strategy as part of secure coding.

Centralize code libraries: Store the project's libraries as well as the rest of the project's code base in a control managed repository. A code repository allows for all of the project's code to be stored in a central place and to manage changes. A security advantage to this is that it makes it easy to backup the entire code base and retrieve it later if the code is damaged [Grembi]. Having the code in one place also allows the development team to assert control over who can access the code base.

Resources

- » Burch, Hal, Long, Fred, Rungta, Raunak, Seacord, Robert C., & Svoboda, David. Specifications for Managed Strings, Second Edition (CMU/SEI-2010-TR-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2010<http://www.sei.cmu.edu/library/abstracts/reports/10tr018.cfm>.
- » "Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today." SAFECode.org. Ed. Stacy Simpson. 8 October 2008. The Software Assurance Forum for Excellence in Code [SAFECode]. 13 July 2010 <http://www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf>.
- » Grembi, Jason. "Secure Processing". Software Assurance Community Resources and Information Clearinghouse. 27 September 2010. 21 October 2010. <https://buildsecurityin.us-cert.gov/swa/presentations_2010_10/01_Monday/grembi/SecureProcessing.pdf>.
- » ISO/IEC TR 24731-1. Extensions to the C Library, — Part I: Bounds-checking interfaces. Geneva, Switzerland: International Organization for Standardization, April 2006.
- » ISO/IEC TR 24731-2. Extensions to the C Library, — Part II: Dynamic Allocation Functions. Geneva, Switzerland: International Organization for Standardization, April 2010.
- » LeBlanc, David. *Integer Handling with the C++ SafeInt Class*. 2004. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp>.
- » Melton, John. "The OWASP Top Ten and ESAPI." John Melton's Weblog: Java, Security and Technology. 3 January 2009. 4 August 2010 <http://www.jtmelton.com/2009/01/03/the-owasp-top-ten-and-esapi/>.
- » "OWASP Enterprise Security API." OWASP.org. 2 July 2010. The Open Web Application Security Project [OWASP]. 4 August 2010 <http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API>.

Secure Coding Principles

Keep Code Small and Simple

The smaller and simpler the code base, the easier it will be to verify the security of the software. The number of flaws in the code that implements high-consequence functions can be reduced significantly by decreasing the size of the source code modules that implement those functions.

Ways to Shrink and Simplify Code

- » Ensure that the software contains only the functions that are required or specified. Adding unnecessary functions adds to the attack surface of the software and raises the likelihood of the software being broken.
- » Break large and/or complex functions into smaller, simpler functions whenever possible. This will make the system easier to understand and document, thus making it easier to verify the security and correctness of the individual component, and of the system as a whole.
- » Build the system so that it has as few interdependencies as possible. This ensures that any process module or component can be disabled or replaced without affecting the operation of the system as a whole.
- » Encapsulate to limit the revealing (leaking) of sensitive information or externally inducing interference.

Consideration should also be given to the tradeoff between size and simplicity. Breaking functions into too many smaller functions can make it difficult to see how the functions work together.

Make Code Backward and Forward Traceable

Where practical, make it easy to trace each requirement to its expression in the design to its manifestation in the code. It should also be possible to derive each requirement and design element from its manifestation in the code.

For more information on developing secure requirements, use cases and misuse case, please reference "Requirements and Analysis for Secure Software" and "Architecture and Design Considerations for Secure Software," both of which can be found in the SwA Pocket Guide Series library.

Code for Reuse and Maintainability

The features that make code secure—simplicity, comprehensibility, and traceability—may also contribute to its reusability and maintainability. If the code can be reused easily, its security properties can be transferred to future projects. If it can be maintained easily, there is less chance of vulnerabilities being added during the maintenance process. One way to code for reuse and maintainability is by creating and adhering to the application guide (discussed above).

Create code that anticipates future events. Ensure that all of the values needed by the software come from a database or an external property file instead of being hard-coded. In Java, for example, consider using input object beans in the method signatures instead of specifying data types and expected values. This allows program functionality to change as needed, and code reuse [Grembi, 242-243].

Resources

» Grembi, Jason. Secure Software Development: A Security Programmer's Guide. Boston: Course Technology, Cengage Learning. 2008.

Follow Secure Coding Standards and/or Guidelines

Select and use secure coding standards and guidelines that explicitly identify safe coding practices and constructs for a given language. These must call out common coding flaws and potentially vulnerable constructs, along with secure alternatives to those problematic flaws/constructs. These standards and guidelines cover both what should be done and what should not be done. For example, the Carnegie Mellon Software Engineering Institute (SEI) published secure coding standards for C, C++, and Java. These guidelines appear in Appendix C:C.4 of "Enhancing the Development Life Cycle to Produce Secure Software" (see "Resources" below).

Though not specifically dealing with security, the Motor Industry Software Reliability Association (MISRA) serves as an example for coding guidelines. MISRA is a collaboration of vehicle manufacturers and companies from related industries that created coding guidelines for C and C++ in an effort to disseminate safety- and reliability-related best practices for embedded electronic systems [MISRA]. Improving the quality of the code is an important part of improving the security of the code.

Once developed, perform conformance testing on the software to determine whether or not coding properly follows the selected secure coding standard. The CERT Program's Source Code Analysis Laboratory (SCALe) offers conformance testing of C language software systems against the CERT C Secure Coding Standard. CERT maintains a registry of certificates for conforming systems. Software systems that successfully pass conformance testing entitle the client to promote the certified software system version with the CERT Conformance Tested seal [Seacord].

Resources

- » CERT Secure Coding Standards. 28 August 2010. Carnegie Mellon University Software Engineering Institute/Computer Emergency Response Team [CERT]. 31 August 2010 <https://www.securecoding.cert.org/>.
- » Seacord, Robert. "Secure Coding". Software Assurance Community Resources and Information Clearinghouse. 1 October 2010. 21 October 2010. < https://buildsecurityin.us-cert.gov/swa/presentations_2010_10/05_Friday/Secure_Coding_SwaForumx.pdf>.
- » United States Government. Department of Homeland Security - Software Assurance Forum Process and Practices Working Group. "Enhancing the Development Life Cycle to Produce Secure Software." The Data & Analysis Center for Software [DACS]. October 2008 Version 2.0. United States Government. Defense Technical Information Center. 24 June 2010 <http://www.thedacs.com/techs/enhanced_life_cycles/>.
- » "What is MISRA?" MISRA.org.uk. The Motor Industry Software Reliability Association [MISRA]. . 4 August 2010 http://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx.

Use Compiler Security Checking and Enforcement

Different compilers are developed for different purposes. The C standard, for example, defines many areas of undefined, unspecified, and implementation-defined behaviors. Many other issues are left to "quality of implementation," meaning that the commercial marketplace determines whether behaviors are acceptable. Frequently, security can be improved by choosing a compiler which implements optional security features or by using methods such as static analysis.

Maximally enable compiler warnings and errors before starting code development. It can be expensive to respond to problems reported later. Require warnings to be eliminated wherever it is practical. If it is not practical to eliminate a warning, document the warning in the code.

Modern development environments, such as Microsoft Visual Studio 2010, include static code analysis functionality that can be employed automatically. Rather than just compiling the code, the Microsoft environment comes with over 200 predefined rules that can be used to check aspects of code against a series of potential weaknesses. The incorporation of automation and development time checking at the individual function level is an essential element in ensuring use of proper development techniques.

Debugging options should be disabled when compiling and linking the production binary executable. For example, some popular commercial operating systems have been reported to contain critical vulnerabilities that enable an attacker to exploit the operating system's standard, documented debug interface. That interface, designed to give the developer control of the program during testing, remains accessible in production systems. It has been exploited by attackers in order to gain control of programs accessed over the network and elevate the attacker's privileges to that of the debugger program.

Some compile-time verification tools leverage type qualifiers. These qualifiers annotate programs so the program can be formally verified as free of recognizable vulnerabilities. Some of these qualifiers are language independent and focus on detecting "unsafe" system calls that must be examined by the developer; other tools detect language-specific vulnerabilities (*e.g., Splint* can do this for C).

In addition, compilers can be modified to detect a maliciously modified stack or data area. A simple form of this protection is the stack canary (a measure first introduced in StackGuard), which is placed on the stack by the subroutine entry code and verified by the subroutine exit code generated by the compiler. If the canary has been modified, the exit code terminates the program with an error.

Many C/C++ compilers can detect inaccurate format strings. For example, the Gnu Compiler Collection supports a C extension that can be used to mark functions that may contain inaccurate format strings, and the */GS* compiler switch in Microsoft's Visual C++ .NET can be used to flag buffer overflows in runtime code.

For C, consider using the as-if infinitely ranged (AIR) integer model, which provides a largely automated mechanism for eliminating integer overflow and truncation and other integral exceptional conditions [Keaton 2010]. The AIR integer model either produces a value equivalent to that obtained using infinitely ranged integers or results in a runtime-constraint violation.

While type and format string checks are useful for detecting simple faults, they are not extensive or sophisticated enough to detect more complex vulnerabilities. Some compile-time tools perform taint analysis, which flags input data as “tainted” and ensures that all such data is validated before allowing it to be used in vulnerable functions. An example is Flayer, an open source taint analysis logic and wrapper [flayer]. This is also built into the Perl language. Other compilers include more extensive logic to perform full program verification and prove complex security properties based on formal specifications. Program verification compilers are most often used to detect flaws and “dangerous” constructs in C and C++ programs and libraries, including constructs that leave the program vulnerable to format string attacks and buffer overflows.

Additional protections that may require compile-time support include compiler randomization of variables and code positions in memory, particularly the randomization of the location of loaded libraries and assembler preprocessors to reduce C and C++ program susceptibility to stack overflows. This often requires both compiler and runtime support.

Resources

- . flayer: Taint analysis and flow alteration tool. 7 October 2009. Google Project Hosting. 27 July 2010 <http://code.google.com/p/flayer/>.
- GNAT Pro High-Integrity Family. AdaCore. July 2010 <http://www.adacore.com/home/gnatpro/development_solutions/safety-critical/>.
- Plum, Thomas & Keaton, David M. “Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool,” 75-81. *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics.* Long Beach, CA, November 2005. U.S. National Institute of Standards and Technology (NIST), 2005.
- Gousset, Mickey. “Static Code Analysis in VS2010”, Visual Studio Magazine, 25 March 2010.

Avoid Security Conflicts Arising Between Native and Non-Native, Passive and Dynamic Code

Applications often rely on code that was written either in another programming language or after the application was originally developed. For example, some non-native, Java applications may rely on native C code to interface directly with hardware. This is a potential vulnerability because an attacker may be able to perform a buffer overflow attack against the native code, even if the Java portions are not vulnerable.

To give another example, an Asynchronous Java and eXtensible Markup Language (AJAX) application may supply dynamically generated JavaScript to the web browser to perform an operation. As a result, the dynamically generated code could have been developed after the original application. This could lead to invalid input to the AJAX application if the dynamically generated code is making different assumptions about the environment and state of the application.

In both cases, it is imperative that applications treat native and dynamic code as potentially un-trusted entities and that developers perform validation on data going to, or coming from, the un-trusted code.

Review Code During and After Coding

The main objective of security code reviews is to discover security flaws and to identify their potential fixes. The test report should provide enough detailed information about software’s possible failure points to enable its developer to classify and prioritize the software’s vulnerabilities based on the likelihood of being exploited by hackers. The application guide (discussed above) should contain a checklist of issues to consider when reviewing code. This will ensure that the code being reviewed will be subject to the documented criteria and not the possible bias of the reviewer.

Tips for a Secure Code Review

- » **When to review**: Code should be reviewed as it is written to locate flaws within individual units/modules after accepting them into the software to be delivered into the software configuration management system (such as GIT or Mercurial). Programmers should also look for flaws in interfaces between units/modules before submitting those units/modules for compilation and linking. Source code analysis and white box testing should be performed as early and as often in the life-cycle as possible. The most effective white box tests are performed on individual modules or functional-process units that can be corrected relatively easily and quickly before they are added into the larger code base. The whole-system code review should focus on the relationships among and interfaces between components.
- » **Peer review:** Programmers should have other programmers on the project review their code. It is easy for programmers to miss their own mistakes [Grembi].
- » **Tool use:** Static, dynamic, and binary analysis tools can be used to find common vulnerabilities. (*e.g.*, Microsoft offers free downloads of FxCop, which is a static analysis tool, BinScope Binary Analyzer, and Mini-Fuzz File Fuzzer.) These tools can produce large amounts of false positives/negatives and should be used in addition to a manual review, not as a replacement.
- » **Reviewer competence**: A manual review is only useful if the reviewers are aware of security issues. At a minimum, they should be aware of the vulnerabilities and mitigations for C/C++ (or whichever language the project is being coded in), databases, web applications, and cryptography [SAFECode 10].
- » **Leveraging test cases:** The programmer should create some general test cases. Testing is not the point of the review but should help point out obvious vulnerabilities [Grembi]. It can also be a chance to test basic abuse cases.
- » **Threat model review**: The secure code review should include a review of the threat model documentation to ensure that identified threats are appropriately handled before code release.
- » **Follow standards**: The secure code review should ensure adherence to all coding standards, including use of secure function libraries and removal of deprecated and banned functions.

What to Look for During a Secure Code Review

- » All components should follow the same framework and the defined coding style [Grembi].
- » Look for developer backdoors, debug commands, hard-coded credentials, sensitive comments in code, overly informative error messages, etc. These are elements added to the source code to make it easier for the developer to alter the software's state while testing. They might be exploitable after the software has been compiled and deployed. The developers should avoid using these elements in the first place unless absolutely necessary.
- » During system execution, look for any unused calls and code that doesn't accomplish anything. They can include calls invoking environment-level or middleware-level processes or library routines that are not expected to be present in the installed target environment.
- » Also look for indicators of malicious code. For example, if the code is written in C, the reviewer might seek out comments that indicate exploit features or portions of code that are complex, hard to follow, or contain embedded assembler code (*e.g.,* the _asm_ feature, strings of hexadecimal or octal characters/values).All identified bugs should be examined for the potential of exploitation. Bugs that can be exploited are security bugs and should be treated specifically prior to code release.

Resources

» BuildSecurityIn White Box Testing resources. 9 September 2009. United States Government. Department of Homeland Security. 28 July 2010 <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box.html>.

» BuildSecurityIn Code Analysis resources. 14 November 2008. United States Government. Department of Homeland Security. 28 July 2010 <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code.html>.

» Chess, Brian and Jacob West. Secure Programming with Static Analysis. Indianapolis, Indiana: Addison-Wesley, 2007.

» "Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today." SAFECode.org. Ed. Stacy Simpson. 8 October 2008. The Software Assurance Forum for Excellence in Code [SAFECode]. 13 July 2010 <http://www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf>.

» Grembi, Jason. "Application Guide." 1 March, 2007. Sterling Connect, LLC.

» Howard, Michael. "A Process for Performing Security Code Reviews." IEEE Security and Privacy Volume 4 Number 4, July/August 2006: 74-79. 28 July 2010 <http://doi.ieeecomputersociety.org/10.1109/MSP.2006.84>.

» Hsu, Francis. "Input validation of client-server Web applications through static analysis". University of California Davis – Department of Computer Science. 14 May 2007. 28 July 2010 <http://mu.cs.ucdavis.edu/~fhsu/papers/web20sp07.pdf>.

» NIST SAMATE - Software Assurance Metrics And Tool Evaluation. July 2005. United States Government. Department of Homeland Security. 28 July 2010 <https://samate.nist.gov/>.

» "OWASP Code Review Project page." OWASP.org. 20 January 2010. The Open Web Application Security Project [OWASP]. 28 July 2010 <http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project>.

» Shostack, Adam. "Security Code Review Guidelines." Adam Shostack's Personal Homepage. May 2000. 28 September 2010 <http://www.homeport.org/~adam/review.html>.

» Wilander, John and Pia Fåk. "Pattern Matching Security Properties of Code using Dependence Graphs". John Wilander's Research Publications (peer-reviewed), 7 November 2005. Linköping University, Department of Computer and Information Science. 27 July 2010 <http://www.ida.liu.se/~johwi/research_publications/paper_cobassa2005_wilander_fak.pdf>.

Secure Coding Practices

SANS Top 25 Error List/OWASP Top 10 List

SANS has published a list of the most common programming errors, and OWASP has done the same for the web application space. These lists represent the actual state of secure coding in current development efforts across many platforms and languages. There is little excuse for a development team to produce software that contains these errors, as they can be specifically checked for and mitigated during the coding process. All members of the development team should be aware of the items on these lists, their causes, mitigations and relative applicability to the current project effort. At code reviews, these specific elements should be verified as being appropriately handled by the code.

Validate and Encode Input

Injection attacks are commonly performed against applications. They can take the form of format string attacks in C or cross-site scripting attacks in web scripting languages. An application that accepts user input and forwards it to the output or some trusted function can be targeted for attack. As a result, it is important to properly validate input to ensure that it meets the application's expectations (*e.g.,* by verifying that the input has a certain length and contains no "special" HTML characters and by securely handling any invalid input). In cases where input validation can allow potentially malicious characters (*e.g.,* "<" in web applications), applications should encode these characters so that they will not be mistaken by other functions or relying applications.

White listing, the preferred approach to input validation, verifies that input conforms to defined acceptable parameters and rejects all input that does not conform. White list validation has been proven to be effective in filtering out input patterns associated with unknown attacks. By contrast, the other approach, black listing, verifies that input matches defined unacceptable parameters associated with suspected malicious input. Black listing accepts all input that is not explicitly identified as "bad." It is useless in protecting against unknown attacks. It can, however, be useful in weeding out some bad input prior to white list filtering. It can also be used as a stopgap for filtering out attack input associated with zero-day vulnerabilities until a patch is available. With black listing, it is important to convert inputs to canonical form. White listing and black listing should be used in combination, with black listing occurring first.

Validate program arguments that define

» Length

» Range

» Format

» Type

Once bad input is detected, it needs to be managed. The best approach is to simply reject bad input, rather than attempting to sanitize it. However, sanitization will sometimes be necessary. When that is the case, all data that is sanitized should be resubmitted for white list validation to ensure the sanitization was effective.

Testing of input validation logic should verify the correctness of the input validation routines. The test scenarios should include submission of both valid and invalid data, and the tester should look for false positives (valid data rejected by input validation) and false negatives (invalid data accepted by input validation). If client-side validation is implemented as the first layer of filtering, it should also be tested with and without JavaScript (JavaScript interprets regular expressions slightly differently than server validation engines do and can generate false positives on the client side and false negatives on the server side.). Security cannot depend on client side testing, as which may quite possibly bypass client side elements. Finally, the tester should also run the application with validation turned off on both the server and client sides. Doing so will reveal how the receipt of invalid data affects other application tiers, and can expose vulnerabilities. In architectures in which input validation is centralized, testing will also reveal whether additional validation may be needed at other points in the system. Ensure that validation cannot be bypassed. In a client/server design (such as a web browser and server) validation at the server is the only validation protecting the server.

Canonicalization

An expression is canonical when it follows the standard format. Canonicalization is the process of reformatting an expression to fit this standard format. SAFECode gives the example that a document called "hello world.doc" can be opened several ways:

» http://www.site.com/hello+world.doc
» http://www.site.com/hello%20world.doc
» http://www.site.com:80/hello%20world.doc [SAFECode 12]

Canonicalization makes sure that expressions are converted to the standard format [SAFECode 12]. Doing this before validation reduces the risk of bypassing filters.

Localization

Localization involves creating variants of a program for use in different countries. Abuse/misuse cases and attack models should be developed for each localized variation of the software. The main problems associated with local variants relate to the potential for creating buffer overflow vulnerabilities due to the longer lengths of resource strings in some languages, the expression of string lengths in bytes versus characters in various character encodings, and the increase in required buffer size that can result from conversions from one encoding to another.

Parsers

Use parsers to manage efforts such as encoding and translation. There are a number of encoding schemes, hex encoding, Unicode encoding, URL encoding, language-based character encoding. One of the challenges is that these encodings can be used together in forms of double encoding, making the canonicalization process a challenge. A series of built in parsers in applications, such as web servers and application servers reduce this challenge to a manageable level. What remains is the issue of where input validation occurs – if it is a pattern-based method of validation, it is important that the check occur after parsers have changed the input to canonical form, but before use. This can be an architectural challenge, and needs careful management.

Additional Resources

The Common Weakness Enumeration (CWE) provides a unified, measurable set of software weaknesses. This compilation enables more effective discussion, description, selection, and use of software security tools and services. The goal is to find weaknesses in source code and operational systems and to better understand and manage software weaknesses related to architecture and design. The best way to mitigate the most egregious exploitable weaknesses in the software development lifecycle is to focus early on the architecture and design phase. With this understanding, managers in development and procurement organizations should question the team responsible for delivering the software exercise practices: how do they better ensure that the architecture and design will sufficiently contribute toward the development of secure software? Some architecture and design security issues are directly related to secure coding, please refer to the "Secure Coding" pocket guide for additional information.

For more details on input validation, see the description and mitigations for CWE 20 "Improper Input Validation" in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide. Also see "Enhancing the Development Life Cycle to Produce Secure Software," Section 7.1.13.1 for details on implementing input validation.

Resources

- » Elgazzar, Mohamed. "Security in Software Localization". Microsoft Global Development and Computing portal. Microsoft. 13 July 2010 <http://www.microsoft.com/globaldev/handson/dev/secSwLoc.mspx>.
- » "Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today." SAFECode.org. Ed. Stacy Simpson. 8 October 2008. The Software Assurance Forum for Excellence in Code [SAFECode]. 13 July 2010 <http://www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf>.

Filter and Sanitize Output and Callouts

Check output that will be passed to external entities (including users, support libraries, and support applications) to ensure conformance to the parameters for allowable output and to ensure that it contains only allowable content. In essence, the same types of checks used for validating input are applied to output before it is released to its intended external recipient. This includes output to the screen and output that goes into other systems. When calling SQL databases, use template calls that automatically escape characters.

A prime example is the function that checks to see if a user is a valid user. If only one record is expected to be returned from the database query, then check for a specific single record return. Just checking to see if the function returns records is flawed as multiple records indicate some form of failure. Likewise, if a database query is to return a single or small number of records, then blocking the return of hundreds or thousands of records via output validation prevents inadvertent database dumping via an application error or the exploitation of a vulnerability.

For more information on securing output, see the mitigations for CWE 116 "Improper Encoding or Escaping of Output" in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide.

Minimize Retention of State Information

The software should retain only the bare minimum of needed state information and should frequently purge data written in cache memory and temporary files on disk. If the software fails, these measures will minimize the likelihood of accidental disclosure of sensitive information, including information about the software itself that can be leveraged by attackers.

However, sometimes the application data must retain state data such as cookies or session data. In these cases, the programmer should be aware of what locations, either in memory or in the file system, can be accessed by an attacker. State data and temporary files should only be placed in areas that cannot be accessed by attackers or anything other than the program. Temporary files should be given names that are difficult for attackers to guess. Also, the cache memory should be purged, and the temporary files should be erased or overwritten as soon as the program terminates [Lewis].

To find out other ways to protect the state data that the software needs to retain, see the mitigations for CWE 642 "External Control of Critical State Data" in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide.

Resources

» Lewis, Richard. "Temporary files security in-depth." Application Security. 12 October 2006. 28 July 2010 <http://secureapps.blogspot.com/2006/10/temporary-files-security-in-depth.html>.

Do Not Allow Unauthorized Privilege Escalation

The programmer should not write logic that enables users or processes to perform escalations of privilege. Attackers can observe a process that attempts to reference another process that has higher privileges than its own. The attackers will interpret such actions on the first process' part as indicating laxness in privilege enforcement and authentication validation by the software system. Processes with higher privileges than the attacker should not be visible to the attacker. If attackers can see the higher privileged process, they can exploit it to escalate their own privileges.

Leverage Security through Obscurity Only as an Additional Deterrence Measure

Security through obscurity measures, such as code obfuscation, use of hidden files, etc., at best provide a weak deterrence against reconnaissance attacks and reverse engineering. While such measures can inconvenience unsophisticated and casual attackers, they should only be used in addition to a robust set of true security measures.

Incorporate Interprocess Authentication

The primary purpose of interprocess authentication is to link the identity and privileges of a human user with those of the application processes that operate on his/her behalf. This allows the software system to make decisions based on the user's (whether human or software entity) security-relevant attributes, such as whether to grant the entity access to certain data or resources. This also allows the system to link a software process' actions to its user's authenticated identity for purposes of accountability. Authentication using Kerberos tickets, Security Assertion Markup Language assertions, SSL/TLS with X.509 certificates or one-time encrypted cookies, secure remote procedure call (RPC) protocols (described below), *etc.,* enables the binding of a human identity with a software process that will act on that human's behalf.

However, new computing models require the ability of autonomous processes to dynamically invoke other autonomous software processes or agents. In turn these agents can dynamically invoke yet more autonomous processes/agents, all of which may be widely distributed throughout the software-intensive system or even belong to other software systems—resulting in a system of systems. Under such conditions, it becomes extremely difficult, if not impossible, to extend the association of a human with each process or agent subsequently invoked by the first process/agent with which the user is associated. Authentication and accountability becomes a matter of keeping track of all the downstream processes/agents that are invoked as part of the need to respond to the original user request or action that originally associated that user with the first process/agent in the chain.

Grid computing initiatives, such as the Globus Grid Security Infrastructure, are defining solutions such as "run-anywhere" single-sign-on authentication of grid agents using SSL/TLS and X.509 certificates [Globus]. Moreover, the emergence of agent-based systems is driving the definition of standards and technical solutions to provide more robust inter-agent authentication and accountability without reference back to a human user.

Resources

» "Overview of the Grid Security Infrastructure." Globus.org. 25 July 2010 <http://www.globus.org/security/overview.html>.

» Yumerefendi, Aydan R. and Jeffrey S. Chase. "Trust but Verify: Accountability for Network Services". Internet Systems and Storage Group: Software architectures for Internet-scale computing. September 2004. The Department of Computer Science at Duke University, Internet Systems and Storage Group. 25 July 2010 <http://issg.cs.duke.edu/publications/trust-ew04.pdf>.

Leverage Attack Patterns

Programmers can use attack patterns to identify specific coding flaws targeted by relevant attacks and ensure that these flaws do not occur in their code. They first need to determine which attack patterns are applicable (*i.e.,* which subset of available attack patterns is relevant given the software's architecture and execution environment) and which technologies are used to implement the software. For example, the *Buffer Overflow* attack pattern would be relevant for a C or C++ program running on native Linux but not for a C# program running on .NET. Based on which attack patterns need to be avoided, programmers should determine which constructs should not appear in their code.

The following example illustrates how a programmer can leverage an attack pattern:

Attack pattern: Simple Script Injection.
Use to: Avoid cross-site scripting vulnerabilities.
Areas of code which this pattern is likely to target: Areas from which output data is sent to the user from an un-trusted source.
How to protect code against this attack pattern: If no countermeasure has already been implemented (*i.e.,* based on an architectural decision to include a self-contained input validator/output filter at the juncture between the server and the client), implement a programmatic countermeasure such as:

1. Convert potentially dangerous characters into their HTML equivalents to prevent the client from displaying un-trusted input that might contain malicious data or artifacts, such as <*script*> tags inserted by an attacker. Examples of such conversions are < *becomes* `&lt;` or > becomes `&gt;`. Third-party Java libraries exist that automatically perform such conversions; JavaScript's *escape()* function also performs similar conversions. Note that such conversions need to be managed carefully in order to avoid potential unintended buffer overflow vulnerabilities that may result from routinely replacing single characters with longer character strings.
2. Implement an input validation filter that filters input based on a white list of allowable characters.

To see the attack patterns that can be used against the most commonly exploited software weaknesses, see the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide. The Common Attack Pattern Enumeration and Classification (CAPEC) is another resource that provides a publicly available catalog of attack patterns along with a comprehensive schema and classification taxonomy.

Implement Encryption and Hashing

Programmers need to encrypt sensitive data (*e.g.*, passwords and session IDs), especially over untrusted channels such as the Internet. Failing to protect sensitive data can lead to attacks (*e.g.*, session hijacking and cross-site scripting) that could result in unauthorized access to the application and data.

Cryptographic functions are an important tool in many software programs, enabling confidentiality and integrity. Cryptography is a demanding science, and cryptographic functions are particularly difficult to create. There are numerous cases of programs that create their own cryptographic functions, only to have them broken later, thus rendering the software insecure. There is no reason for this failure as secure, vetted cryptographic function libraries exist for all major languages. The use of these libraries and the use of non-deprecated cryptographic functions is an easy task, easier than developing the functions on your own. The following are a few recommendations regarding algorithms:

- Use Transport Layer Security (TLS) for web applications.
- Avoid Message-Digest 5 (MD5) and Secure Hash Algorithm 1 (SHA-1); instead, use SHA-256/512 and SHA-3 for hashing.
- Avoid Data Encryption Standard (DES); instead, use Advanced Encryption Standard (AES) or Triple DES.

Programmers should also code applications to allow easy switching of the hash and encryption algorithms in case the algorithm is broken. It is important to note that the Federal Information Processing Standard (FIPS) 140-2 is applicable to all Federal agencies that use cryptographic-based security systems and provides lists of approved cryptographic functions and validated cryptographic modules.

Disable Debugging Tools Prior to Deployment

Core dumps are only acceptable as a diagnostic tool during testing. Applications should be configured upon deployment to turn off their ability to generate core dumps when they fail during operational use. Instead of dumping core when the program fails, the program's exception handler should post to a secure log the appropriate data before the program exits. In addition, if possible, the size of the core file should be configured as 0 (zero) (*e.g.*, using *setrlimit* or *ulimit* in UNIX); this will further prevent the creation of core files.

Secure Memory and Cache Management

The following are strategies to ensure that programmers create software that manages memory and its caches securely.

Limit Persistent Memory Caching

Much of today's software is written to maximize performance through extensive use of persistent memory caching. The problem with persistent memory is that the longer data remains in memory, the more opportunity there is for that data to be inadvertently or intentionally disclosed if a failure causes a core dump or otherwise allows the content of memory to become directly accessible. The problem arises because memory is not subject to the access control protections of the operating-system-level file system, or in a database application, the database management system.

For example, when using entity beans on a Java Platform, Enterprise Edition (Java EE) server, data can be stored on the server with either container-level persistence or bean-level persistence. In both cases, the program that processes and caches the data needs to provide secure cache management capabilities that can accommodate all of the simultaneous processes the program has to multitask (*e.g.,* receipt and handling of requests, management of sessions, reading of data from a database or file system). The developer can force a program that uses Enterprise Java Beans to write data to persistent non-volatile storage after each transaction instead of making the data persistent in memory; the tradeoff is one of performance (fewer database accesses means better performance) versus security (less memory persistence means less opportunity for attackers to access sensitive data not protected by file system or database access controls). In multi-user programs, the amount of overhead required to securely manage persistent cache is also affected by the number of users who simultaneously access the program; the higher the amount of data that must be cached, the lower the amount of memory that will be available to other processes.

If writing a program in which memory will be persistent, the developer should ensure that the length of persistence is configurable, so that it will be purged as frequently as the administrator or user desires. Ideally, the program will also provide a command that allows the administrator or user to purge the memory at will. Regardless of the length of memory persistence when the program is running, data should always be purged from memory as soon as the program shuts down. For high-consequence and trusted processes, the program should not retain the memory beyond the completion of the process; it should be purged when the process completes.

Ideally, extremely sensitive data, such as authentication tokens and encryption keys will never be held in persistent memory. However, persistent memory cannot be avoided in commercial off-the-shelf (COTS) (including open-source software) components. In these cases, if the component is likely to store sensitive data in persistent memory, the developer should leverage the cache management and object reuse capabilities of the operating system and, if there is one, the database management system to overwrite each persistent memory location with random bits seven times (the number of times considered sufficient for object reuse). When using a COTS component that makes memory persistent, if the data the component will store in memory is likely to be sensitive, consider hosting the component on a Trusted Processor Module (TPM) to isolate its persistent memory from the rest of the system.

Allocate Memory and Other Resources Carefully

Minimize the computer resources made available to each process. For example, for software that will run on Unix, use *ulimit(), getrlimit(), setrlimit(), getrusage(), sysconf(), quota(), quotactl(),* and *quotaon()* (also *pam_limits* for pluggable authentication module processes) to limit the potential damage that results when a particular process fails or is compromised and to help prevent denial-of-service (DoS) attacks on the software.

If the software is a web server application or web service, the developer should set up a separate process to handle each session and limit the amount of central processing unit (CPU) time that each session in each process is allowed to use. Doing so will prevent any attacker request that causes excessive memory or CPU cycles from interfering with tasks beyond its own session. It will also make it difficult to create DoS attacks by spawning multiple sessions.

Memory locations for cache buffers should not be contiguous with executable stack/heap. Whenever possible, the stack should be non-executable. Data Execution Prevention (DEP), a security feature for Windows operating systems, provides an easy way to ensure that writable memory cannot be executed [Pop]. DEP can help block a malicious program in which a virus or other type of attack has injected a process with additional code and then tries to run the injected code. On a system with DEP, execution of the injected code causes an exception. Software-enforced DEP can help block programs that take advantage of exception-handling mechanisms in Windows.

To avoid return-to-libc attacks, where the return address of the program is overwritten to execute what the attacker wants, the system code should be moved to different points in the memory. Address Space Layout Randomization (ASLR), another security feature for Windows operating systems, moves the executable, the libraries, and other important parts of the code, to random points in the memory every time the system is booted [Pop]. This makes it harder for exploit code to operate predictably.

Resources

- » "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003". Microsoft Support. 26 September 2006. Microsoft. 17 November 2010 <http://support.microsoft.com/kb/875352>.
- » Howard, Michael. "Address Space Layout Randomization in Windows Vista". Michael Howard's Web Log: A Simple Software Security Guy at Microsoft! 26 May 2006. MSDN Blogs. 17 November 2010 <http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx>.
- » Pop, Alin Rad. "DEP/ASLR Implementation Progress in Popular Third-party Windows Applications". Secunia.com. 29 June 2010. Secunia. 17 November 2010 <http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf>.

Secure Error and Exception Handling

Software has to be survivable to be secure. Survivable software contains more error- and exception-handling functionality than program functionality. All error- and exception-handling routines should be made with two goals in mind:

1. Handle faults to prevent the software from entering an insecure state.
2. To ensure that the software can continue operating in a degraded manner until a threshold is reached that triggers an orderly, secure termination of the software's execution.

Developers who "reactively extend" native exception handlers (*e.g.,* by pasting exception classes into source code after an error or fault has been observed in the program's testing or operation) need to be careful not to rely solely on this approach to exception handling. Otherwise the result will be a program that fails to capture and handle any exceptions that have not yet been thrown during testing or operation. Exception handling should be proactively designed to the greatest extent possible, through careful examination of code constraints as they occur during the concept and implementation phases of the life cycle.

The developer should list all predictable faults (exceptions and errors) that could occur during software execution and define how the software will handle each of them. In addition, the developer ought to address how the software will behave if confronted with an unanticipated fault or error. In some cases, potential faults may be preempted in the design phase, particularly if the software has been subjected to sufficiently comprehensive threat modeling. In others, developers could preempt additional faults during pseudo-coding by cautiously examining the logical relationships between software objects and developing "pseudo" exception handling routines to manage these faults.

Table 1, below, lists several common software errors and suggests remediation methods for those errors at the design and implementation levels.

Table 1 -- Software Errors and Suggested Remediation Methods

Expected Problem	How Software Should Handle the Problem
Input contains anomalous content.	IMPLEMENTATION: Restrict the possible input whenever possible. Also validate all input.
The execution environment differs significantly from the environment for which the software was designed.	DESIGN: Recognize all explicit and implicit assumptions COTS and OSS components have about their environment. Design to mitigate vulnerabilities created by mismatch between component assumptions and actual environment. Design also to minimize external exposure of component/environment interfaces.
There are errors in the results returned by called functions.	DESIGN: Design for resilience. IMPLEMENTATION: Anticipate all likely errors and exceptions, and implement error and exception handling to explicitly address those errors/exceptions.
The software contains vulnerabilities that were not mitigated or detected before the software was deployed.	DESIGN: Include measures that isolate un-trusted, suspicious, and compromised components, and that constrain and recover from damage. IMPLEMENTATION: Measures that reduce exposure of un-trusted and vulnerable components to externally sourced attack patterns (*e.g.,* using wrappers, input filters, *etc.).*

The exception-handling block should log when and why an exception was thrown; auditors can review this log later. The exception-handling block should also be written to include messaging code that will automatically send an alert to the system administrator when an exception requires human intervention.

Above all, the software should never throw exceptions that allow it to crash, dump core memory, or expose its caches, temporary files, and other transient data exposed. To find ways to prevent data being exposed from error messages, see the mitigations for CWE 209 "Error Message Information Leak" in the Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses pocket guide.

Practices for implementing effective error and exception handling are described below.

Integrate Anomaly Awareness

In most distributed software systems, components maintain a high level of interaction with each other. Abnormal behavior includes inaction (*i.e.,* lack of response) in a particular component for an extended period of time, and generation of messages that do not follow prescribed protocols. All components should be designed or retrofitted to recognize abnormal behavior patterns that indicate possible DoS attempts. Place this capability in software developed from scratch and retrofit acquired or reused components (*e.g.,* by adding anomaly-detection wrappers to monitor the component's behavior and report detected anomalies).

Early detection of the anomalies that are typically associated with a DoS can make mitigation possible before a full DoS occurs. Techniques include containment, graceful degradation, and automatic fail-over. While anomaly awareness alone cannot prevent a widespread DoS attack, it can effectively handle isolated DoS events in individual components, as long as detected abnormal behavior patterns correlate with anomalies that can be handled by the software as a whole.

Incorporate Runtime Error Checking and Safety Enforcement

Apply runtime protections to prevent buffer overflows in binaries executed under a particular operating system or to analyze compiled binaries in terms of dynamic runtime security. Apply security wrappers and content validation filters to open-source software code and binary executables to minimize the exposure of their vulnerabilities. By and large, security wrappers and validation filters are used to add content (input or output) filtering logic to programs that don't have that logic "built in." First, the wrappers intercept and analyze input to or output from the "wrapped" program. Next they detect and then remove, transform, or isolate content that is suspected of being malicious (*e.g.,* malicious code) or that contains unsafe constructs, such as very long data strings (associated with buffer overflows) and command strings associated with escalation of privilege. As a general rule, security wrappers and content validation filters must be custom developed.

Use Event Monitors

Use an event—error or anomaly—monitor to ensure that anomalies or errors are detected as soon after the causal event as possible. Early and accurate detection will enable isolation and diagnosis before erroneous data can propagate to other components. Time and memory availability usually limit the number of program self-checks that can be implemented. At a minimum, checks should occur for all security-critical states. Use risk analysis to establish the basis for defining the optimal contents and locations of each check.

An attacker will quickly move to erase any evidence of an attack. Because of this, in the event of an attack, logs collected by the event monitor should be promptly sent to a remote server for further analysis.

Monitor checks should be non-intrusive; they should not corrupt the process or data being checked. In addition, the developer should take particular care when coding logic for monitoring and checking to avoid including exploitable flaws.

The effectiveness of event monitors is based on the correctness of assumptions about the structure of the program being monitored and the anomalies and errors that are considered likely or unlikely in the program. However as with all assumptions, these may be invalidated under certain conditions.

What to Avoid

Just as there are best practices for secure coding, there are also worst practices. Listed below are some of the more basic programming errors. Please note that this is not a comprehensive list. Please refer to the references in "Resources" below for additional examples of bad coding practices.

Do Not Use Deprecated or Insecure Functions

Using deprecated, insecure, or obsolescent functions should be avoided in favor of equivalent, more-secure functions. In C language programming, for example, deprecated functions are defined by the C99 standard and Technical Corrigenda. The gets() function was deprecated by Technical Corrigendum 3 to C99 and eliminated from C1X. Alternatives to gets() include fgets() and gets_s(), of which gets_s() is better at replicating gets().

The International Organization for Standardization/International Electrotechnical Commission Technical Report (ISO/IEC TR) 24731 defines alternative versions of C standard functions that are designed to be safer replacements for existing functions. For example, ISO/IEC TR 24731 Part I (24731-1) defines the strcpy_s(), strcat_s(), strncpy_s(), and strncat_s() functions as replacements for strcpy(), strcat(), strncpy(), and strncat(), respectively. The ISO/IEC TR 24731-1 functions were created by Microsoft to help retrofit its existing, legacy code base in response to numerous, well-publicized security incidents over the past decade. These functions were subsequently proposed to the international standardization working group for the programming language C (ISO/IEC Joint Technical Committee 1 (JTC1)/Subcommittee 22 (SC22)/Working Group 14 (WG14)).

References for Other Languages

Like C/C++, other languages keep track of functions that have been deprecated. Java for instance retains a Javadocs list of deprecated functions (The list for Java Platform SE6 is available at http://download.oracle.com/javase/6/docs/api/).

Another example is Perl which provides deprecation information in the manual pages that come with it. *perlfunc* describes Perl's functions and says if the function is experimental, deprecated, discouraged, or removed (see the *perlpolicy* manual page to see what these terms mean for Perl). There is also *perldelta*, which will say if any functions have been deprecated in the latest release.

Regardless of which language is used, avoid all commands and functions known to contain exploitable vulnerabilities or otherwise unsafe logic. None of the obscure, unfamiliar features of a language should be used unless the required functionality cannot be achieved any other way. However even in those circumstances, the unfamiliar features should be carefully researched to ensure the developer understands all of their security implications.

Resources

» Rationale for TR 24731 Extensions to the C Library Part I: Bounds-checking interfaces. http://www.open-std.org/JTC1/SC22/WG14/www+/docs/n1173.pdf.

» See guideline: INT05-C. Do not use input functions to convert character data if they cannot handle all possible inputs [CERT C Secure Coding Standard 2010].

» See guideline: FIO12-C. Prefer setvbuf() to setbuf() [CERT C Secure Coding Standard 2010].

» See guideline: FIO07-C. Prefer fseek() to rewind() [CERT C Secure Coding Standard 2010].

Do Not Use Filenames in an Improper Manner

Most programs deal with files (*e.g.*, opening, reading, creating a file). Naturally, dealing with files in an improper way can lead to serious security vulnerabilities such as race conditions, unauthorized disclosure or modification of the file's contents, and so on. One way to cause race conditions is by referring to a file by its name more than once in the same program. Don't check to see if the file exists before opening it; instead, attempt to open files directly, then react to errors.
Once a file is opened, it should be given a file handle and referred to only by that file handle for the rest of the program. Referring to a file by its name more than once in a program gives the attacker a chance to alter the file between references and trick the program into trusting information that should not be trusted [Graff and van Wyk 111].

Referring to a file with a relative filename can cause unauthorized disclosure or modification of a file's contents. Filenames should be fully qualified, meaning it starts with a device name such as "C:\Documents and Settings\Desktop\text.txt." Using a relative filename gives an attacker an opportunity to change the working directory and access files that the attacker should not have access to [Graff and van Wyk 110].

Resources

» Graff, Mark G., and Kenneth R. Van Wyk. Secure Coding Principles & Practices. Sebastopol, CA: O'Reily. June 2003.

Questions to Ask Software Developers

The following are questions managers could ask their development teams or acquirers of software services could ask their suppliers. These questions highlight the major coding considerations for assuring secure applications. Intended to raise awareness of the content of this Guide, they are not a complete set of questions. A more comprehensive set of questions can be found in the Pocket Guides for “Software Assurance in Acquisition and Contract Language” and “Software Supply Chain Risk Management & Due-Diligence.”

- What input and output protection mechanisms are implemented within the application?
- What measures are taken to ensure that users only have access to that which they are allowed to view?
- How are user passwords stored?
- How are connection strings stored?
- What cryptographic functions are implemented within the application, and what considerations went into selecting this over others?
- What tasks are included in each phase of the software development life cycle to identify and address security concerns?
- Who is involved in the code review process and how is it reviewed?
- If any, which third-party libraries are used, and how is it checked to ensure it does not contain malicious code or vulnerabilities?
- What type of auditing is done to keep track of changes made to database records?
- What language is used and why?
- What are the strengths and weaknesses of the language?
- Describe the security aspects of the development process. Provide your development policy and application guides.

Conclusion

This pocket guide compiles best practices and resources on the subject of secure coding. The material and resources provided in the guide can be used as a starting point for making a secure and repeatable coding process. As stated at the beginning, it is not the purpose of this Guide to provide a comprehensive description of every security situation that can arise while developing code. Rather, the Guide offers basic direction for establishing good secure coding processes and procedures. Developers should review the referenced documents to extend their knowledge of secure coding.

For a more interactive way of learning secure coding see the mini-courses offered by the SANS Software Security Institute. CERT and SEI also offer a “Secure Coding in C” course through the Carnegie Mellon University Open Learning Initiative. (Register with the course key: seccode.) Google and OWASP also offer teaching tools — Gruyere (previously called Jarlsberg) and WebGoat, respectively — that allows users to learn about common secure coding mistakes. Microsoft has made a significant effort to share with all developers, lessons learned, tools and techniques to improve secure coding efforts. The industry group SAFECode also has a host of company-agnostic materials. These secure coding lessons that can be implemented over a wide range of development efforts. There exists a whole host of valuable information on how to write code securely, the challenge is in adopting this information into a specific development process.

The Software Assurance Pocket Guide Series is developed in collaboration with the SwA Forum and Working Groups and provides summary material in a more consumable format. The series provides informative material for SwA initiatives that seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development, acquisition, and deployment of trustworthy software products. It should be understood by the reader that these pocket guides are not an authoritative source, they simply reference and point to authoritative sources. Together, these activities will enable more secure and reliable software that supports mission requirements across enterprises and critical infrastructure.

For additional information or contribution to future material and/or enhancements of this pocket guide, please consider joining any of the SwA Working Groups and/or send comments to Software.Assurance@dhs.gov. SwA Forums are open to all participants and free of charge. Please visit https://buildsecurityin.us-cert.gov for further information.

No Warranty

Reprints

Software Assurance (SwA) Pocket Guide Series

SwA is primarily focused on software security and mitigating risks attributable to software; better enabling resilience in operations. SwA Pocket Guides are provided; with some yet to be published. All are offered as informative resources; not comprehensive in coverage. All are intended as resources for 'getting started' with various aspects of software assurance. The planned coverage of topics in the SwA Pocket Guide Series is listed:

SwA in Acquisition & Outsourcing

I. Software Assurance in Acquisition and Contract Language
II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

I. Integrating Security in the Software Development Life Cycle
II. Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
III. Software Security Testing
IV. Requirements & Analysis for Secure Software
V. Architecture & Design Considerations for Secure Software
VI. Secure Coding
VII. Security Considerations for Technologies, Methodologies & Languages

SwA Life Cycle Support

I. SwA in Education, Training & Certification
II. Secure Software Distribution, Deployment, & Operations
III. Code Transparency & Software Labels
IV. Assurance Case Management
V. Assurance Process Improvement & Benchmarking
VI. Secure Software Environment & Assurance Ecosystem
VII. Penetration Testing throughout the Life Cycle

SwA Measurement & Information Needs

I. Making Software Security Measurable
II. Practical Measurement Framework for SwA & InfoSec
III. SwA Business Case

SwA Pocket Guides and related documents are freely available for download via the DHS NCSD Software Assurance Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa.