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

Fundamental Practices for Secure Software Development

2ND EDITION

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

February 8, 2011

Editor Stacy Simpson, SAFECode

Authors

Mark Belk, Juniper Networks
Matt Coles, EMC Corporation
Cassio Goldschmidt, Symantec Corp.
Michael Howard, Microsoft Corp.
Kyle Randolph, Adobe Systems Inc.
Mikko Saario, Nokia
Reeny Sondhi, EMC Corporation
Izar Tarandach, EMC Corporation
Antti Vähä-Sipilä, Nokia
Yonko Yonchev, SAP AG

Foreword

In 2008, the Software Assurance Forum for Excellence in Code (SAFECode) published the first version of this report in an effort to help others in the industry initiate or improve their own software assurance programs and encourage the industry-wide adoption of what we believe to be the most fundamental secure development methods. This work remains our most in-demand paper and has been downloaded more than 50,000 times since its original release.

However, secure software development is not only a goal, it is also a process. In the nearly two and a half years since we first released this paper, the process of building secure software has continued to evolve and improve alongside innovations and advancements in the information and communications technology industry. Much has been learned not only through increased community collaboration, but also through the ongoing internal efforts of SAFECode's member companies. This 2nd Edition aims to help disseminate that new knowledge.

Just as with the original paper, this paper is not meant to be a comprehensive guide to all possible secure development practices. Rather, it is meant to provide a foundational set of secure development practices that have been effective in improving software security in real-world implementations by SAFECode members across their diverse development environments.

It is important to note that these are the "practiced practices" employed by SAFECode members, which we identified through an ongoing analysis of our members' individual software security efforts. By bringing these methods together and sharing them with the larger community, SAFECode hopes to move the industry beyond defining theoretical best practices to describing sets of software engineering practices that have been shown to improve the security of software and are currently in use 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.

Though expanded, our key goals for this paper remain—keep it concise, actionable and pragmatic.

What's New

This edition of the paper prescribes new and updated security practices that should be applied during the Design, Programming and Testing activities of the software development lifecycle. These practices have been shown to be effective across diverse development environments. While the original also covered Training, Requirements, Code Handling and Documentation, these areas were given detailed treatment in SAFECode's papers on security engineering training and software integrity in the global supply chain, and thus we have refined our focus in this paper to concentrate on the core areas of design, development and testing.

The paper also contains two important, additional sections for each listed practice that will further increase its value to implementers—Common Weakness Enumeration (CWE) references and Verification guidance.

CWE References

SAFECode has included CWE references for each listed practice where applicable. Created by MITRE Corp., CWE provides a unified, measurable set of software weaknesses that can help enable more effective discussion, description, selection and use of software security practices. By mapping our recommended practices to CWE, we wish to provide a more detailed illustration of the security issues these practices aim to resolve and a more precise starting point for interested parties to learn more.

Verification

A common challenge for those managing software security programs is the need to verify that development teams are following prescribed security practices. SAFECode aims to address that challenge with this new edition. Wherever possible, we have included methods and tools that can be used to verify whether a practice was applied. This is an emerging area of work and SAFECode hopes to use community feedback to further bolster its guidance in this area.

Software vendors have both a responsibility and a business incentive to ensure software security. SAFECode has collected and analyzed the secure development methods currently in use among its members in order to provide others in the industry with highly actionable advice for improving software security. This is a living document and we plan to continue to update it as the industry and practices evolve. Thus, SAFECode encourages feedback and suggestions as to how we can continue to improve this paper's usefulness to readers.

SAFECode has published a series of papers on software supply chain integrity that aim to help others understand and minimize the risk of vulnerabilities being inserted into a software product during its sourcing, development and distribution.

The software integrity controls discussed in the papers are used by major software vendors to address the risk that insecure processes, or a motivated attacker, could undermine the security of a software product as it moves through the links in the global supply chain. The controls aim to preserve the quality of securely developed code by securing the processes used to source, develop, deliver and sustain software and cover issues ranging from contractual relationships with suppliers, to securing source code repositories, to helping customers confirm the software they receive is not counterfeit.

Copies of *The Software Supply Chain Integrity Framework: Defining Risks and Responsibilities for Securing Software in the Global Supply Chain* and *Overview of Software Integrity Controls: An Assurance-Based Approach to Minimizing Risks in the Software Supply Chain* are available at www.safecode.org.

SAFECode encourages all software developers and vendors to consider, tailor and adopt these practices into their own development environments. The result of efforts like these will not only benefit industry through a more secure technology ecosystem, but also provide a higher level of end-user confidence in the quality and safety of software that underpins critical operations in governments, critical infrastructure and businesses worldwide.

Table of Contents

Foreword ii

What's New ii

CWE References iii

Verification iii

Introduction 2

Secure Design Principles 2

Threat Modeling 2

CWE References 5

Verification 5

Resources 6

Use Least Privilege 7

CWE References 8

Verification 8

Resources 9

Implement Sandboxing 10

CWE References 10

Verification 10

Resources 11

Secure Coding Practices 12

Minimize Use of Unsafe String and Buffer Functions 12

CWE References 13

Verification 14

Resources 15

Validate Input and Output to Mitigate Common Vulnerabilities 15

CWE References 17

Verification 17

Resources 18

Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets 19

CWE References 20

Verification 20

Resources 21

Use Anti-Cross Site Scripting (XSS) Libraries 22

CWE References 24

Verification 24

Resources 26

Use Canonical Data Formats 27

CWE References 27

Verification 28

Resources 28

Avoid String Concatenation for Dynamic SQL Statements 29

CWE References 29

Verification 30

Resources 31

Eliminate Weak Cryptography 32

CWE References 33

Verification 34

Resources 35

Use Logging and Tracing 37

CWE References 37

Verification 38

Resources 38

Testing Recommendations 39

Determine Attack Surface 39

Use Appropriate Testing Tools 39

Perform Fuzz / Robustness Testing 40

Perform Penetration Testing 41

CWE References 41

Verification 42

Resources 42

Technology Recommendations 44

Use a Current Compiler Toolset 44

CWE References 45

Verification 45

Resources 46

Use Static Analysis Tools 47

CWE References 49

Verification 49

Resources 49

Summary of Practices 50

Moving Industry Forward 51

Acknowledgements 51

Introduction

A review of the secure software development processes used by SAFECode members reveals that there are corresponding security practices for each activity in the software development lifecycle that can improve software security and are applicable across diverse environments. The examination of these vendor practices reinforces the assertion that software security must be addressed throughout the software development lifecycle to be effective and not treated as a one-time event or single box on a checklist. Moreover, these security methods are currently in practice among SAFECode members, a testament to their ability to be integrated and adapted into real-world development environments.

The practices defined in this document are as diverse as the SAFECode membership, spanning cloud-based and online services, shrink-wrapped and database applications, as well as operating systems, mobile devices 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.

Secure Design Principles

Threat Modeling

The most common secure software design practice used across SAFECode members is Threat Modeling, a design-time conceptual exercise where a system's dataflow is analyzed to find security vulnerabilities and identify ways they may be exploited. Threat Modeling is sometimes referred to as "Threat Analysis" or "Risk Analysis."

Proactively understanding and identifying threats and potential vulnerabilities early in the development process helps mitigate potential design issues that are usually not found using other techniques, such as code reviews and static source analysis. In essence, Threat Modeling identifies issues before code is written—so they can be avoided altogether or mitigated as early as possible in the software development lifecycle. Threat Modeling can also uncover insecure business logic or workflow that cannot be identified by other means.

Rather than hope for an analysis tool to find potential security vulnerabilities after code is implemented, it's more efficient for software development teams to identify potential product vulnerability points at design time. This approach enables them to put in place defenses covering all possible input paths and institute coding standards to help to control the risk right from the beginning. It is worth noting that an analysis tool lacks knowledge of the operating environment in which the system being analyzed executes.

By their nature, systemic architectural issues are more costly to fix at a later stage of development. Thus, Threat Modeling can be considered a cost-efficient, security-oriented activity, because fixing issues early in the process may be as easy as changing an architecture diagram to illustrate a change to a solution yet to be coded. In contrast, addressing similar issues after coding has begun could take months of re-engineering effort if they are identified after code was committed, or even a major release or a patch release if an issue was identified even later by customers in the field.

Leveraging the full benefits of Threat Modeling when designing systems can be challenging as software designers and architects strive to identify all possible issues and mitigate them before moving forward. This can be difficult to achieve, so the focus must be on the high-risk issues that can be identified at design time. In addition, Threat Modeling results should be continuously updated as design decisions change and added threats may become relevant, and threats may be mitigated during development or by virtue of documentation or clearly visible use case limitations.

Threat Modeling can be done at any time in the system's lifecycle, but to maximize effectiveness the process should be performed as early in the development process as possible. Distinct software development methodologies will have different points where system design may change: in a traditional "waterfall" development model, Threat Modeling would be performed when the design is relatively well established but has not yet been finalized, and in the Agile model, the activity could occur during initial design or be a recurring activity during each iteration or sprint—when the design is most likely to undergo change.

The process of Threat Modeling begins with the identification of possible and commonly occurring threats. Different SAFECode practitioners have adopted different approaches to the task of enumerating threats against the design being analyzed:

- "STRIDE" – this methodology classifies threats into 6 groups: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege. Threat Modeling is executed by looking at each component of the system and determines if any threats that fall into these categories exist for that component and its relationships to the rest of the system.
- "Misuse cases" – The employment of misuse cases helps drive the understanding of how attackers might attack a system. These cases should be derived from the requirements of the system, and illustrate ways in which protective measures could be bypassed, or areas where there are none. For example, a misuse case involving authentication would state "By successively entering login names, an attacker can harvest information regarding the validity (or not) of such login names."
- "Brainstorming" – if an organization does not have expertise in building threat models, having a security-oriented discussion where the

designers and architects evaluate the system is better than not considering potential application weaknesses at all. Such “brainstorming” should not be considered a complete solution, and should only serve as a stepping stone to a more robust Threat Modeling exercise.

- “Threat library” – a format that makes threat identification more accessible to non-security professionals. Such a library must be open to changes to ensure it reflects the evolving nature of threats. Publicly available efforts like CWE (Common Weakness Enumeration—a dictionary of software weakness types), OWASP (Open Web Application Security Project) Top Ten and CAPEC (Common Attack Pattern Enumeration and Classification that describes common methods of exploiting software) can be used to help build this library. Use of a Threat library can be a quick way to take advantage of industry security knowledge (helping teams that lack sufficient knowledge themselves) or combine elements of other Threat Modeling methods (such as linking a threat to misuse cases and a STRIDE classification).

Once identified, each threat must be evaluated and mitigated according to the risk attached to it (using a risk rating system such as Common Vulnerability Scoring System (CVSSv2), for example), the resources available, the business case and the system requirements. This will help prioritize the order in which threats should be addressed during development, as well as the costs involved in the mitigation. At times, this will drive changes in design to enable less costly mitigations. Even without available mitigations or design changes introduced, a complete Threat Model provides a good way to measure and manage security risk in applications.

The end result of a Threat Modeling exercise may vary, but it will certainly include an annotated diagram of the system being evaluated, as well as a list of the associated threats (mitigated and not).

It has been observed in some cases that Threat Modeling as part of recurring activities in the Software Development Lifecycle helps to drive a culture that accepts security as an integral aspect of software design—the benefit is cumulative, with later iterations building on the experience of earlier ones.

Different approaches offer varying requirements of prior security expertise in order to achieve good results, and it is possible to choose the one that better suits the situation at hand, and later on change to another approach based on the improving awareness to security in the involved participants.

As a conceptual exercise, Threat Modeling will highly benefit from close communication since having all those responsible present in one location can lead to lively, results-generating discussion. However, geographically dispersed teams will still be able to conduct Threat Modeling exercises using the many means of communication at their disposal, from remote presence setups to spreadsheets and diagrams sent over email. The speed of the exercise may vary, but there are no specific

negative impacts to the end result if the exercise becomes a question-answer discussion using email, for example.

Tools are available that support the Threat Modeling process with automated analysis of designs and suggestions for possible mitigations, issue-tracking integration and communication related to the process. Some practitioners have honed their Threat Modeling process to the point where tools are used to automate as much of it as possible, raising the repeatability of the process and providing another layer of support with standard diagramming, annotation, integration with a threat database and test cases, and execution of recurring tasks.

CWE References

Much of CWE focuses on implementation issues, and Threat Modeling is a design-time event. There are, however, a number of CWEs that are applicable to the threat modeling process, including:

- CWE-287: Improper authentication is an example of weakness that could be exploited by a Spoofing threat
- CWE-264: Permissions, Privileges, and Access Controls is a parent weakness of many Tampering, Repudiation and Elevation of Privilege threats
- CWE-311: Missing Encryption of Sensitive Data is an example of an Information Disclosure threat
- CWE-400: (uncontrolled resource consumption) is one example of an unmitigated Denial of Service threat

Verification

A comprehensive verification plan is a direct derivative of the results of the Threat Model activity. The Threat Model itself will serve as a clear roadmap for verification, containing enough information so that each threat and mitigation can be verified.

During verification, the Threat Model and the mitigated threats, as well as the annotated architectural diagrams, should also be made available to testers in order to help define further test cases and refine the verification process. A review of the Threat Model and verification results should be made an integral part of the activities required to declare code complete.

An example of a portion of a test plan derived from a Threat Model could be:

Threat Identified	Design Element(s)	Mitigation	Verification
Session Hijacking	GUI	Ensure random session identifiers of appropriate length	Collect session identifiers over a number of sessions and examine distribution and length
Tampering with data in transit	Process A on server to Process B on client	Use SSL to ensure that data isn't modified in transit	Assert that communication cannot be established without the use of SSL

Resources

References:

- OWASP; "Open Web Application Security Project"; http://www.owasp.org
- CWE; "Common Weakness Enumeration"; http://cwe.mitre.org
- CAPEC; "Common Attack Pattern Enumeration and Classification"; http://capec.mitre.org
- CVSSv2; "Common Vulnerability Scoring System"; http://www.first.org/cvss/

Presentations:

- AND-304: Threat Modeling: Lessons Learned and Practical Ways To Improve Your Software; RSA Conference 2010; Dhillon & Shostack

Books, Articles and Reports:

- The Security Development Lifecycle; Chapter 9, "Stage 4: Risk Analysis"; Microsoft Press; Howard & Lipner
- 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); http://iac.dtic.mil/iatac/download/security.pdf
- Software Security; Chapter 2, "A Risk Management Framework"; McGraw; Addison-Wesley; 2006.
- Security Mechanisms for the Internet; Bellovin, Schiller, Kaufman; http://www.ietf.org/rfc/rfc3631.txt
- Capturing Security Requirements through Misuse Cases; Sindre and Opdahl; http://folk.uio.no/nik/2001/21-sindre.pdf
- Software Security; Chapter 8, "Abuse Cases"; McGraw; Addison-Wesley; 2006.

Tools / Tutorials:

- The Microsoft SDL Threat Modeling Tool; http://www.microsoft.com/security/sdl/getstarted/threatmodeling.aspx

Use Least Privilege

The concept of executing code with a minimum set of privileges is as valid today as it was 30 years ago when it was described in Saltzer and Schroeder's seminal paper, "The Protection of Information in Computer Systems." The concept of least privilege is simple, but it can be hard to achieve in some cases. Even though "least privilege" means different things in different environments, the concept is the same:

"Every program and every user of the system should operate using the least set of privileges necessary to complete the job."

Least privilege is important because it can help reduce the damage caused if a system is compromised. A compromised application running with full privileges can perform more damage than a compromised application executing with reduced privileges. The value of operating with reduced privileges cannot be stressed enough.

The concept of privilege varies by operating system, development technologies and deployment scenarios. For example:

- Most mobile platforms will force all non-operating system code to run in a sandbox running with minimal privilege, but developers should still only select the privileges or permissions required for the application to execute correctly. For example:
    - Android requires applications to describe the permissions needed by the application in the application's AndroidManifest.xml file.
    - Windows Phone 7 uses WMAppManifest.xml to describe application capabilities.
    - Symbian applications can have capabilities assigned to them.
    - iOS applications have the concept of "entitlements."
- .NET applications can describe required permissions in the app.manifest file.
- Java can do likewise in the policy file named java.policy.
- Windows applications and services run under an account (a Security Identifier [SID]) that is granted group membership and privileges.
- Linux applications and daemons run under an account that has implicit privileges.
- Some Linux distributions (e.g. MeeGo) use capabilities derived from the now-defunct POSIX 1003.1e draft (also referred to as POSIX.1e).
- Some Linux distributions (e.g.; Fedora and RedHat) use SELinux, which provides extensive technologies including SIDs and labels.
- Some Linux distributions (e.g.; Ubuntu and Suse) use AppArmor, which supports some POSIX 1003.1e draft capabilities and supports application profiles.

- Grsecurity is a set of patches for Linux that provide, amongst other security tools, role-based access control (RBAC) mechanisms.

In short, privileges, capabilities and entitlements determine which sensitive operations can be performed by applications and users. In the interests of security, it is imperative that sensitive operations be kept to a minimum.

There are two development aspects of least privilege that must be considered. The first is making sure that the application operates with minimum privileges and the second is to test the application fully in a least privilege environment. Developers are notorious for building and smoke-testing applications using full privilege accounts, such as root or members of the administrators group. This can lead to problems during deployment, which are usually conducted in low-privilege environments. It is strongly recommended that all developers and testers build and test applications using least privilege accounts.

The second point of consideration is to thoroughly test the application in a least privilege environment to shake out least-privilege related bugs. It is recommended that the application under test be subject to a complete test pass and all security-related issues noted and fixed.

Finally, a least privilege environment must include tamper proof configuration, otherwise applications or users might be able to grant more trusted capabilities.

CWE References

Like sandboxing, the core CWE is the following:

- CWE-250: Execution with Unnecessary Privileges

Verification

Verifying an application is running with least privilege can be subjective, but there are some tools that can provide details to help an engineer understand which permissions and privileges are granted to a running process:

- In Windows, Application Verifier will issue "LuaPriv" warnings if potential least privilege violations are detected at runtime.
- For Windows Phone 7, the Windows Phone Capability Detection Tool can help determine what the permission set should be for a Windows Phone 7 application.

Least privilege is typically enforced in applications via configurable user or code permissions. Therefore, performing regular audits or reviews of the default permissions can be an effective means toward ensuring least privilege in secure code. The review can be based on a software specification, outlining user roles or the functions of supplementary components, or via a post-implementation validation of the software, for example, with integration tests.

Resources

Books, Articles and Reports:

- The Protection of Information in Computer Systems; Saltzer, Schroeder; http://www.cs.virginia.edu/~evans/cs551/saltzer/
- nixCraft; Linux Kernel Security (SELinux vs AppArmor vs Grsecurity); Gite; http://www.cyberciti.biz/tips/selinux-vs-apparmor-vs-grsecurity.html
- SAP Developer Network; Integrated Identity and User Management; http://www.sdn.sap.com/irj/sdn/go/portal/prtroot/com.sap.km.cm.docs/library/netweaver/netweaver-developers-guide-2004s/SAP%20NetWeaver%20Developer%27s%20Guide%202004s/IUAM%20Further%20Information.ca
- Authorizations in SAP Software: Design and Configuration; Lehnert, Bonitz & Justice; SAP Press; 2010.

Presentations:

- Linux Capabilities: Making Them Work; Linux Symposium 2008; Hallyn, Morgan; http://ols.fedoraproject.org/OLS/Reprints-2008/hallyn-reprint.pdf

Tools / Tutorials:

- Android Manifest.permission; http://developer.android.com/reference/android/Manifest.permission.html
- MSDN Library; Application Manifest File for Windows Phone; http://msdn.microsoft.com/en-us/library/ff769509(v=VS.92).aspx
- MSDN Library; How to: Use the Windows Phone Capability Detection Tool; http://msdn.microsoft.com/en-us/library/gg180730(VS.92).aspx
- MSDN Library; Windows Application Verifier; http://msdn.microsoft.com/en-us/library/dd371695(VS.85).aspx

Implement Sandboxing

While the concept of "sandboxing" processes is not new, the industry has seen an increase in interest in the topic since the first version of this paper was written.

Running a process in a user's session on many popular operating systems usually implies that the process has all of the privileges and access rights to which the user is entitled. No distinction is made between what a user's web browser should have access to and what their word processing software should have access to. This model has three risks of abuse of those privileges:

a. Unrestricted execution of arbitrary native code achieved via memory corruption bugs

b. Abuse of functionality using the privileges available to the user

c. Executing arbitrary code from within a managed code (C#, Java, Python, Ruby etc) runtime environment

Using a managed language, such as C# or Java, defends against the first scenario by managing memory on behalf of the application. Managed runtimes also have their own sandboxes to defend against the second scenario using policy-driven code access security. When switching to a managed language is not an option, such as in large legacy code bases, sandboxing offers an alternative mitigation by utilizing operating system security features to restrict the abilities of a sandboxed process.

Features provided by operating systems to support sandboxing functionality include:

- Process-level memory isolation
- Integrity Levels on Windows
- Dropping process privileges
- Disabling high-privilege user accounts used by the process
- Running each application as a unique user
- Permission Manifests
- File system 'jails'

Applications that are installed on a large number of systems (>1 million, for example) and process untrusted data from the Internet are highly encouraged to implement sandboxing. In addition, applications that are installed as plugins to high-risk applications like browsers should work within the host application's sandbox.

Many current mobile platforms run all applications in a sandboxed environment by default.

CWE References

There is one parent CWE that relates directly to sandboxing:

- CWE-265: Privilege / Sandbox Issues

Verification

- Ensure that all ingredients provided by the platform for a sandbox are implemented correctly

by reviewing the resources below for the target platform. One missing ingredient can render the entire sandbox protection ineffective.

- Review the attack surface that is available from within the sandbox. This can be accomplished using tools like SandKit, which enumerates all resources that are accessible from within the sandbox. Validate that each item found performs adequate input validation and authorization checks.
- Review the sandbox policy to ensure the least amount of access necessary is granted. For example, review an Android application's androidmanifest.xml for granted permissions that are too relaxed.

Resources

Books, Articles and Reports:

- Practical Windows Sandboxing – Part 1; Leblanc; http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx
- Inside Adobe Reader Protected Mode – Part 1 – Design; McQuarrie, Mehra, Mishra, Randolph, Rogers; http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html

Resources (continued)

Tools / Tutorials:

- Chromium Sandbox Design Document; http://www.chromium.org/developers/design-documents/sandbox
- OS X Sandboxing Design; http://www.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design
- iOS Application Programming Guide: The Application Runtime Environment; http://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/RuntimeEnvironment/RuntimeEnvironment.html#//apple_ref/doc/uid/TP40007072-CH2-SW44l
- Android Security and Permissions; http://developer.android.com/guide/topics/security/security.html
- The AndroidManifest.xml file; http://developer.android.com/guide/topics/manifest/manifest-intro.html
- SandKit; http://s7ephen.github.com/SandKit/

Secure Coding Practices

In this section, the focus shifts to the low-level development-related practices used by SAFECode members.

Minimize Use of Unsafe String and Buffer Functions

Memory corruption vulnerabilities, such as buffer overruns, are the bane of applications written in C and C++. An analysis of buffer overrun vulnerabilities over the last 10 years shows that a common cause of memory corruption is unsafe use of 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.

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

Development engineers should be instructed to avoid using these classes of function calls. Using tools to search the code for these calls helps verify that developers are following guidance and helps identify problems early in the development cycle. Building the execution of these tools into the "normal" compile/build cycle relieves the developers from having to take "special efforts" to meet these goals.

It is important to be aware of library- or operating system-specific versions of these function classes. For example, Windows has a functional equivalent to strcpy called lstrcpy and Linux has a memcpy equivalent called bcopy, to name a few, and these too should be avoided.

Some example replacement functions include:

Unsafe Function	Safer Function
strcpy	strcpy_s
strncpy	strncpy_s
strcat	strcat_s
strncat	strncat_s
scanf	scanf_s
sprintf	sprintf_s
memcpy	memcpy_s
gets	gets_s

Developers using C++ should consider using the classes built into the standard language library to manipulate buffers and strings. For example, rather than using strcpy or strncpy in C++, developers should use std::string objects.

The memcpy function deserves special mention because many developers believe it is safe. It is safe when used correctly, but if an attacker controls the number of bytes to copy, or the developer incorrectly calculates the buffer size, then the function becomes insecure. SAFECode believes that developers should move away from using memcpy in favor of memcpy_s as the latter forces the developer to think about the maximum destination buffer size.

Automatic use of safer functions

Both Microsoft Visual C++ and GNU gcc offer an option to migrate some buffer-copying function calls to safer calls if the destination buffer size is known at compile time. Consider adding the following definitions to the respective compiler options:

Visual C++: -D_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES=1

gcc: -D_FORTIFY_SOURCE=2 -O2

Some SAFECode members note that using these options can make code review more complex because the resulting object code differs from the source code. However, the benefit of using these options is high as in many cases over 50 percent of insecure functions are migrated to safer function calls in legacy code for very little engineering effort.

CWE References

There are many CWE entries that related to memory- and buffer-related issues, including:

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

Verification

The following tools and techniques can be used to verify this practice is used.

Tool or Technique	Outcome
banned.h	No function deprecation warnings when compiling with this header
Coverity	No warnings from the "OVERRUN_STATIC" checker
Fortify SCA 360	C/C++: Buffer Overflow None of the following warnings: C/C++: Format String C/C++: Buffer Overflow (Off-by-One) C/C++: Buffer Overflow (Signed Comparison) C/C++: Out-of-Bounds Read C/C++: Out-of-Bounds Read (Off-by-One) C/C++: Out-of-Bounds Read (Signed Comparison)
Klocwork	No warnings from the "NNTS", "NNTS.TAINTED", "SV.STRBO.GETS", "SV.STRBO.UNBOUND_COPY", "SV.STRBO.UNBOUND", "SPRINTF" checkers
Microsoft Visual C++	None of the following warnings: C4996 The following require the code to be compiled with /analyze: C6029 C6053 C6057 C6059 C6200 C6201 C6202 C6203 C6204
RATS	No "Severity: High" warnings

Resources

Books, Articles and Reports:

- Please Join Me in Welcoming memcpy() to the SDL Rogues Gallery; http://blogs.msdn.com/b/sdl/archive/2009/05/14/please-join-me-in-welcoming-memcpy-to-the-sdl-rogues-gallery.aspx

Presentations:

- strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation; USENIX 99; Miller, de Raadt; http://www.usenix.org/events/usenix99/millert.html

Tools / Tutorials:

- banned.h; http://www.microsoft.com/downloads/en/details.aspx?FamilyID=6aed14bd-4766-4d9d-9ee2-fa86aad1e3c9
- Strsafe.h; http://msdn.microsoft.com/en-us/library/ms647466(VS.85).aspx
- SafeStr; https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/271-BSI.html

Validate Input and Output to Mitigate Common Vulnerabilities

Checking the validity of incoming data and rejecting non-conformant data can remedy the most common vulnerabilities that lead to denial of service, data or code injection and misuse of end user data. In some cases, checking data validity is not a trivial exercise; however, it is fundamental to mitigating risks from common software vulnerabilities.

Checking the validity of outgoing data can remedy many web-based vulnerabilities, such as cross site scripting, as well as mitigate information leakage issues.

Data enter and exit an application in the form of a byte stream, which is then interpreted into variables with specific parameters for length and data type. Input validation refers to checking data validity before it is processed by the application, whereas output validation refers to validating application data after it is processed, with the purpose of matching the expectations of its intended recipient. For successful data validation, the variable's contents should be validated according to the following guidelines:

- Input variable must be checked for existence and for conformance to specified data lengths.
- Data must be normalized, or transformed into its simplest and shortest representation. Also referred to as canonicalization. This topic is discussed in more detail in "Use Canonical Data Formats" on page 27.

- Data must be checked for conformance with data types specified by the application and its output recipients.
- For fields with clear value ranges, data must be checked for conformance with a specified value range.
- A whitelist filter should be applied to limit input to allowed values and types. For data where defining such a whitelist is not possible, the data validation should be performed against a blacklist of disallowed values and data types.

A whitelist is a list or register of data elements and types that are explicitly allowed for use within the context of a particular application. By contrast, a blacklist is a list or register of data elements and types that are explicitly disallowed for use within a particular application. Whitelisting typically constrains the application inputs to a pre-selected list of values, whereas blacklisting gives more freedom and rejects only the banned data elements and/or types. Applications should not rely solely on using blacklists as there are often many ways around the list using various escaping mechanisms. This is especially true for web-based applications.

Another approach with greater flexibility is to use data validating libraries for input and output validation and cleanup during development. Such data validating libraries are available for almost all programming languages and application platforms. To be effective, this approach requires disciplined application of data validation to all input and output. The implementation of data validation libraries should be supported by an explicit requirement in a secure development standard or specification document.

In some user applications types, notably web-based applications, validating and/or sanitizing output is critical in mitigating classes of attacks against user applications, arising from vulnerabilities such as cross-site scripting, HTTP response splitting and cross-site request forgery.

For applications running on a remote server and consumed over the network from a user client, data validation should take place on the server. Implementing data validation within the user client can be bypassed and is discouraged. If data validation at the user client can't be avoided, it should be associated with data validation at the server application and the corresponding error handling.

Data validation should also not be neglected for applications that exchange data with other applications without user interaction, particularly for applications that expose functions via remotely callable interfaces—either via proprietary or standardized protocols such as SOAP, REST or others. Interfaces that accept text and structured XML data, can use regular expressions or string comparisons for validation against data type descriptors.

Last but not least, nontransparent and harder-to-validate binary or encoded data should at minimum be checked for data length and field validity. Additionally, the source of the binary data may be verified with the use of digital signatures. The use of digital signatures as a data validation method should, in general, be deployed for data exchanges with integrity protection requirements, such as the exchanges in banking transactions. In these cases, signature validation should be the very first check that is applied.

CWE References

Input and output validation is often the parent issue that leads to many classes of vulnerability such as XSS, buffer overruns and cross-site request forgery. CWE captures the high-level nature of this weakness in a number of CWEs including the following:

- CWE-20: Improper Input Validation
- CWE-183: Permissive Whitelist
- CWE-184: Incomplete Blacklist
- CWE-625: Permissive Regular Expression

Verification

An effective way to verify this practice is to look for the existence and use of validation methods within the application. The specific methods should be described in secure development guidelines, requiring the use of libraries or manual input and output verification and when they should be used.

The verification of the proper application of the recommended methods can be performed via standardized QA methods such as code reviews or automated code scanning tools. Verification should be performed during the active application development phase, ideally in close collaboration with interface definitions during application design phases.

Resources

Books, Articles and Reports:

- Writing Secure Code 2nd Ed; Chapter 10, All Input is Evil!; Howard, LeBlanc; Microsoft Press.
- Protecting Your Web Apps: Two Big Mistakes and 12 Practical Tips to Avoid Them; Kim, Skouis; SANS; http://www.sans.org/reading_room/application_security/protecting_web_apps.pdf
- JavaWorld; Validation with Java and XML Schema, Part 1; Mclaughlin; http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html?page=1

Tools / Tutorials:

- SAP Developer Network Secure Programming Guides; http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/334929d6-0a01-0010-45a9-8015f3951d1a
- Input and Data Validation; ASP.NET; http://wiki.asp.net/page.aspx/45/input-and-data-validation/
- Data Validation; OWASP; http://www.owasp.org/index.php/Data_Validation
- Flash Validators; http://code.google.com/p/flash-validators/
- Struts; OWASP; http://www.owasp.org/index.php/Struts
- Java Data Validation - Swing Components; http://www.java2s.com/Code/Java/Swing-Components/Data-Validation.htm

Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets

There are three types of integer issues that can result in security vulnerabilities such as buffer overflows:

- Overflow and underflow
- Signed versus unsigned errors
- Data truncation

These integer issues can occur during arithmetic, assignment, cast, type conversion, comparison, shift, boolean and binary operations.

It's important to note that this issue can apply to all programming languages, not just C and C++.

The proper solution is to use robust integer datatypes, such as the ones provided in the SafeInt library, which force robust handling of all integer operations. When this solution is not feasible to implement, the following best practices are recommended:

- Use unsigned integers (such as DWORD and size_t) for array indexes, pointer offsets, and buffer sizes.
- Use unsigned integers for while, do, and for loops. An integer overflow can occur in the loop during increment and decrement operations of the index variable. These overflows may cause either an infinite loop or reading/writing a large number of bytes from a buffer.
- Do not use signed integers for arguments to memory allocation functions or array offsets; use unsigned integers instead.
- Check that the number of elements expected (e.g.; number of bytes in a request) is no larger than a predetermined value that is smaller than the largest amount of memory the application should allocate.

Other general best practices for robust handling of integers:

- Pay attention to the assumptions about sign and size of data types in and across different languages, platforms, compilers, or managed to unmanaged code. For example, a size_t is a different type depending on the platform you use. A size_t is the size of a memory address, so it is a 32-bit value on a 32-bit platform, but a 64-bit value on a 64-bit platform.
- Compile code with the highest possible warning level, such as /W4 when using Visual C++ or -Wall when using gcc.
- When available, enable compiler features to detect integer issues, such as -ftrapv in gcc.
- Catch exceptions for detected integer issues if they are provided by the platform or language. Some languages and platforms may need a special directive to throw exceptions for detected integer issues. For example, use the checked keyword in C#.

- It is not necessary to use robust integer operations when the integers involved cannot be manipulated by an attacker. Assumptions like this must be evaluated regularly as the software evolves.

CWE References

- CWE-129: Improper Validation of Array Index
- CWE-190: Integer Overflow or Wraparound
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-680: Integer Overflow to Buffer Overflow
- CWE-805: Buffer Access with Incorrect Length Value

Verification

A blend of actions is recommended to verify that safe integer arithmetic has been implemented:

- Review static analysis output for arithmetic issues. Results vary widely by static analysis tool.
- Review compiler output resulting from a compilation with a high warning level enabled, such as '/W4'. Results vary by compiler. In general, compilers are typically more effective at identifying signed/unsigned mismatches and truncation issues than overflows and underflows. Examples of warnings related to integer issues include C4018, C4389 and C4244.
- Investigate all use of pragmas that disable compiler warnings about integer issues. Comment them out, re-compile and check all new integer-related warnings.
- Develop fuzzing models that exercise inputs used for pointer arithmetic, such as arguments used for payload size and array offset. Also, have the models exercise boundary conditions, such as -1 and 0xFFFFFFFF.
- Manually review the code for functions that allocate memory or perform pointer arithmetic. Make sure that the operands are bounded into a small and well-understood range.

The following tools and techniques can be used to verify this practice is used.

Tool or Technique	Outcome
Coverity	No warnings from the "OVERRUN_DYNAMIC", "MISRA_CAST", "NEGATIVE_RETURNS", "REVERSE_NEGATIVE", "TAINTED_SCALAR" checker
Fortify SCA 360	C/C++: Buffer Overflow (Off-by-One) C/C++: Format String C/C++: Out-of-Bounds Read C/C++: Out-of-Bounds Read (Off-by-One) C/C++: Integer Overflow C/C++: Buffer Overflow C/C++: Buffer Overflow (Signed Comparison) C/C++: Out-of-Bounds Read (Signed Comparison)
Klocwork	No warnings from the "SV.TAINTED.ALLOC_SIZE", "ABV.TAINTED Buffer", "SV.TAINTED.CALL.INDEX_ACCESS", "SV.TAINTED.INDEX_ACCESS" checkers
RATS	No "Severity: High" warnings

Resources

Books, Articles and Reports:

- Phrack; Basic Integer Overflows; Blexim; http://www.phrack.org/issues.html?issue=60&id=10#article
- Safe Integer Operations; Plakosh; Pearson Education; https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding/312-BSI.html?layoutType=plain
- MSDN Library; Integer Handling with the C++ SafeInt Class; LeBlanc; http://msdn.microsoft.com/en-us/library/ms972705
- The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities; Dowd, McDonald, Shuh; ISBN: 978-0321444424.

Tools / Tutorials:

- MSDN Library; Reviewing Code for Integer Manipulation Vulnerabilities; Howard; http://msdn.microsoft.com/en-us/library/ms972818

Use Anti-Cross Site Scripting (XSS) Libraries

This section is a web-specific variant of "Validate input and output to mitigate common vulnerabilities" above.

Cross Site Scripting (XSS) stands for a class of vulnerabilities in applications that allow the injection of active scripting data into script-enabled application screens. XSS-based attacks most often target script-interpreting web clients, generally web browsers. XSS attacks occur by maliciously injecting script into an application that fails to validate incoming and outgoing data. A successfully conducted attack that exploits XSS vulnerabilities can lead to serious security violations such as user privilege escalation, user impersonation, code injection, user client hijacking and even background propagation of XSS based attacks.

A cross site scripting attack is typically executed in the following steps:

1. Attacker identifies input fields into a web-based application, which lack input validation and are reused to generate static or dynamic output in a subsequent script-enabled application screen. Attackers may use visible or hidden input fields in the input pages, or input parameters exchanged via web application URLs.
2. The attacker misuses the identified input fields to inject active scripts in the application flow. The script code may be delivered directly in the input field, remotely via a custom URL or based on a previous injection. A variant of XSS, DOM-based XSS, can also misuse input for legitimate client-side scripts to execute malicious scripts on the user client side.
3. Once the user browser displays the static or dynamically-generated HTML, generated from the misused input field, the malicious script is identified as such by the user browser and automatically executed. With its automated browser-side execution, the script runs under the browser privilege of the user client and is able to access and misuse private user data that is shared with the browser.

As a defense-in-depth measure, XSS issues can be avoided by validating all output that may include untrusted user client-originating input. The large number of input and output fields in a typical web application, however, makes manual validation of every field impractical. As an alternative to manual validation, the use of anti-XSS libraries, or web UI frameworks with integrated XSS protection, can minimize the developer's efforts by correctly validating application input and outputs. Anti-XSS libraries are available for most web application platforms, where exposure to XSS attacks is highest. The resources section contains a list of the most popular ones; further references are available from the web platform vendor's support documentation.

Generally, anti-XSS measures must be built in to software applications when the following conditions are present:

1. Application accepts input from users

2. The input is used for dynamic content generation, or is displayed to users in a subsequent script-enabled application screen.

While XSS protections can be used to a large extent by applying output validation techniques, input validation addresses the root cause of the XSS vulnerabilities. As a general rule, both must always be used in conjunction with each other. In addition to the techniques outlined in section “Validate Input and Output to mitigate common vulnerabilities,” the basic development rules to avoid XSS vulnerabilities, as well as criteria for anti XSS library selection, are as follows:

- Constrain Input:
  - Define a codepage (such as charset = ISO-8859-1) to narrow down problematic characters.
  - Filter meta-characters based on their intended interpreter (e.g. HTML client, web browser, file system, database, etc.) Used alone, this practice is not secure; therefore filtering meta-characters should be considered an extra defensive step.
- Normalize input, or bring it to a specified form before its validation.
- Validate all user input at the server:
  - Against a whitelist, to accept only known unproblematic characters or data types
  - If users are allowed to enter a URL within the input field, restrict the domain of the URL and permit only the selection of approved URLs.
- Encode all web applications outputs so that any inserted scripts are prevented from being transmitted to user browsers in an executable form.
  - Use HTML meta elements to clearly identify the character encoding in the output document.
  - Depending on the output context and the encoding used, convert problematic meta-characters originating from user input, for example in HTML < to *&lt;* , > to *&gt;* , and “ to *&quot;*
  - Wherever feasible, encode the whole page displayed to the user to plain HTML. This measure has to be used carefully as it also deactivates capabilities for dynamic web page content and customizations.

In addition, most of the current web browsers offer options for deploying user client-side protection measures, via browser plug-ins, or as in integral part of the browser UI rendering engine. By adding an “HTTPOnly” flag to client-side cookies, user clients can also be instructed to limit cookie use and make cookies unavailable to access from an active script or one embedded in the browser objects (Java applet, ActiveX control, etc.). Anti-virus solutions can also validate to some extent user client-side application inputs and detect attacks. For local

applications with script-enabled UIs, placing the UIs in a sandboxed file system location can also help to reduce the available attack surface.

Client-side protection measures against XSS are, however, web browser or client platform specific and their consistent use by users can't be relied upon. Therefore, client-side protection against XSS should not be considered a replacement for server side protection that uses input and output validation methods or anti-XSS libraries.

CWE References

The following CWE is relevant to XSS issues:

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

There are many child CWEs that relate to web vulnerabilities:

- CWE-81: Improper Neutralization of Script in an Error Message Web Page
- CWE-82: Improper Neutralization of Script in Attributes of IMG Tags in a Web Page
- CWE-83: Improper Neutralization of Script in Attributes in a Web Page
- CWE-84: Improper Neutralization of Encoded URI Schemes in a Web Page
- CWE-85: Doubled Character XSS Manipulations
- CWE-86: Improper Neutralization of Invalid Characters in Identifiers in Web Pages
- CWE-87: Improper Neutralization of Alternate XSS Syntax

Verification

Verification follows the basic rules laid out in the section "Validate Input and Output to Avoid Common Security Vulnerabilities." Detailed strategies for mitigating XSS vulnerabilities are also listed in the referenced CWE.

The following methods can be used to find XSS issues:

- Automated code scanning tools with application data flow analysis capabilities
- Code scanning or reviews to verify the application of anti-XSS libraries or proper application input and output validation methods

The following tools and techniques can be used to verify this practice is used.

Tool or Technique	Outcome
Fortify SCA 360	None of the following warnings: .NET: Cross-Site Scripting (Persistent) .NET: Cross-Site Scripting (Reflected) .NET: Cross-Site Scripting (Poor Validation) Java: Cross-Site Scripting (DOM) Java: Cross-Site Scripting (Persistent) Java: Cross-Site Scripting (Reflected) Java: Cross-Site Scripting (Poor Validation) JavaScript: Cross-Site Scripting (DOM) PHP: Cross-Site Scripting (Persistent) PHP: Cross-Site Scripting (Reflected) PHP: Cross-Site Scripting (Poor Validation) Python: Cross-Site Scripting (Persistent) Python: Cross-Site Scripting (Reflected) Python: Cross-Site Scripting (Poor Validation) SQL: Cross-Site Scripting (Persistent) SQL: Cross-Site Scripting (Reflected) SQL: Cross-Site Scripting (Poor Validation) VB/VB.NET: Cross-Site Scripting (Persistent) VB/VB.NET: Cross-Site Scripting (Reflected) VB/VB.NET: Cross-Site Scripting (Poor Validation) ColdFusion: Cross-Site Scripting (Persistent) ColdFusion: Cross-Site Scripting (Reflected) ColdFusion: Cross-Site Scripting (Poor Validation)
Klocwork	No warnings from the “NNTS “, “NNTS.TAINTED”, “SV.STRBO.GETS”, “SV.STRBO.UNBOUND_COPY”, “SV.STRBO.UNBOUND”,_”SPRINTF” checkers

Resources

References:

- Apache Wicket; http://wicket.apache.org/
- OWASP Top 10 2010, Cross Site Scripting; http://www.owasp.org/index.php/Top_10_2010-A2
- Wikipedia Entry; http://en.wikipedia.org/wiki/Cross_site_scripting
- IE 8 XSS Filter; http://www.microsoft.com/windows/internet-explorer/features/safer.aspx

Tools / Tutorials:

- OWASP Enterprise Security API; Interface Encoder; http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Encoder.html
- OWASP PHP AntiXSS Library; http://www.owasp.org/index.php/Category:OWASP_PHP_AntiXSS_Library_Project
- Microsoft Web Protection Library; http://www.codeplex.com/AntiXSS
- OWASP Reviewing Code for Cross-site scripting; http://www.owasp.org/index.php/Reviewing_Code_for_Cross-site_scripting
- Mozilla Content Security Policy; http://people.mozilla.org/~bsterne/content-security-policy/index.html
- OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet; http://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet
- SAP Developer Network, Secure Programming Guides; http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/334929d6-0a01-0010-45a9-8015f3951d1a
- MSDN Library; Microsoft Anti-Cross Site Scripting Library V1.5: Protecting the Contoso Bookmark Page; Lam; http://msdn.microsoft.com/en-us/library/aa973813.aspx
- Microsoft Code Analysis Tool .NET (CAT.NET) v1 CTP-32 bit; http://www.microsoft.com/downloads/en/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en

Use Canonical Data Formats

Where possible, applications that use resource names for filtering or security defenses should use canonical data forms. Canonicalization, also sometimes known as standardization or normalization, is the process for converting data that establishes how various equivalent forms of data are resolved into a “standard,”“normal,” or canonical form. For example, within the context of a windows file path, the data file ‘Hello World.docx’ may be accessible by any one of the following paths:

“C:\my files\Hello World.docx”

“C:\my files\Hello World.docx” (same as above, but the ‘o’ in docx is a Cyrillic letter, U+043E)

“c:\my files\hello worLD.docx”

c:\myfile~1\hellow~1.doc

“C:/my files/Hello World.docx”

“\\?\c:\files\hello.pdf”

“%homedrive%\my files\Hello World.docx”

“\\127.0.0.1\C$\my files\Hello World.docx”

“C:\my files\.\..\my files\Hello World.docx”

“\ :-) \..\my files\\\\Hello World.docx”

Besides the use of numerous canonical formats, attackers on the web often take advantage of rich encoding schemes available for URL, HTML, XML, JavaScript, VBScript and IP addresses when attacking web applications. Successful attacks may allow for unauthorized data reading, unauthorized data modification or even denial of service, thus compromising confidentiality, integrity and availability respectively.

Canonical representation ensures that the various forms of an expression do not bypass any security or filter mechanisms. Best design practices suggest all decoding should be executed first using appropriate APIs until all encoding is resolved. Next, the input needs to be canonicalized. Only then can authorization take place.

CWE References

The CWE offers many examples of canonicalization issues, including:

- CWE-21: Pathname Traversal and Equivalence Errors
- CWE-22: Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
- CWE-35: Path Traversal: ‘.../...//’
- CWE-36: Absolute Path Traversal
- CWE-37 Path Traversal: ‘/absolute/pathname/here’
- CWE-38 Path Traversal: ‘\absolute\pathname\here’
- CWE-39 Path Traversal: ‘C:dirname’
- CWE-40 Path Traversal: ‘\\UNC\share\name\’

Verification

Few tools can find real canonicalization issues, but automated techniques can find areas where path traversal weaknesses exist. However, tuning or customization may be required to remove or de-prioritize path-traversal problems that are only exploitable by the software's administrator or other privileged users.

Examples of automated tests include adding extra path details (such as path traversal characters), changing case and using escaped characters at random when running stress tests that exercise file access. This could be considered a form of directed fuzz testing.

The following tools and techniques can be used to verify this practice is used.

Tool or Technique	Outcome
Coverity	No warnings from the "TAINTED_STRING" checker
Fortify SCA 360	ColdFusion: Path Manipulation C/C++: Path Manipulation .NET: Path Manipulation Java: Path Manipulation PHP: Path Manipulation Python: Path Manipulation VB/VB.NET: Path Manipulation
Veracode	None for the aforementioned CWE weakness Tests used: Automated Static

Resources

Books, Articles and Reports:

- Writing Secure Code 2nd Ed.; Chapter 11 "Canonical Representation Issues"; Howard & Leblanc; Microsoft Press.
- Hunting Security Bugs; Chapter 12 "Canonicalization Issues"; Gallagher, Jeffries & Lanauer; Microsoft Press.

Tools / Tutorials:

- OWASP ESAPI Access Reference Map API; http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/AccessReferenceMap.html
- OWASP ESAPI Access Control API; InterfaceAccess Controller; http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/AccessController.html
- Microsoft KnowledgeBase; How to Programmatically Test for Canonicalization Issues with ASP.NET; http://support.microsoft.com/kb/887459
- MSDN Library; PathCanonicalize Function (Win32); http://msdn.microsoft.com/en-us/library/bb773569(VS.85).aspx
- MSDN Library; .Net Framework 4 URI class; http://msdn.microsoft.com/en-us/library/system.uri.aspx
- SAP Developer Network Secure Programming Guides; http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/334929d6-0a01-0010-45a9-8015f3951d1a

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. Except in very rare instances, string concatenation should not be used to build SQL statements. Common misconceptions include the use of stored procedures, database encryption, secure socket layer (SSL), and removal and duplication of single quotes as ways to fix SQL injection vulnerabilities. While some of those techniques can hinder an attack, only the proper use of SQL placeholders or parameters can 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 as well as to understand the importance of avoiding disclosing database information in error messages.

Proper database configuration is a vital defense in depth mechanism and should not be overlooked: ideally, only selected stored procedures should have execute permission and they should provide no direct table access. System accounts servicing database requests must be granted the minimum privilege necessary for the application to run. If possible, the database engine should be configured to only support parameterized queries.

SQL injection flaws can often be detected using automated static analysis tools. False positives may arise when automated static tools cannot recognize when proper input validation was performed. Most importantly, false negatives may be encountered when custom API functions or third-party libraries invoke SQL commands that cannot be verified because the code is not available for analysis.

Successful SQL injection attacks can read sensitive data, modify data and even execute operating system level commands.

CWE References

There is one major CWE:

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Verification

OWASP offers pertinent testing advice to uncover SQL injection issues (see Resources). Various tools can help detect SQL injection vulnerabilities:

Tool or Technique	Outcome
Microsoft CAT.NET (using SQL Injection checks)	No "A SQL injection vulnerability was found ..." warnings
Microsoft Visual Studio Code Analysis	No CA2100 warnings
Microsoft FxCop (Microsoft.Security category)	No CA2100 warnings
W3AF (sqli and blindSqli plugins)	No warnings
Fortify SCA 360	ColdFusion: SQL Injection C/C++: SQL Injection .NET: SQL Injection .NET: SQL Injection (Castle Active Record) .NET: SQL Injection (Linq) .NET: SQL Injection (NHibernate) .NET: SQL Injection (Subsonic) Java: SQL Injection Java: SQL Injection (JDO) Java: SQL Injection (Persistence) Java: SQL Injection (Ibatis Data Map) JavaScript: SQL Injection PHP: SQL Injection Python: SQL Injection SQL: SQL Injection VB/VB.NET: SQL Injection
Veracode	None for the aforementioned CWE weakness Tests used: Automated Static, Automated Dynamic, Manual

Resources

References:

- OWASP; SQL Injection; http://www.owasp.org/index.php/SQL_Injection

Books, Articles and Reports:

- Giving SQL Injection the Respect it Deserves; Howard; http://blogs.msdn.com/sdl/archive/2008/05/15/giving-sql-injection-the-respect-it-deserves.aspx
- Unixwiz.net; SQL Injection Attacks by Example; Friedl; http://www.unixwiz.net/techtips/sql-injection.html

Tools / Tutorials:

- OWASP; Guide to SQL Injection; http://www.owasp.org/index.php/Guide_to_SQL_Injection
- OWASP; Testing for SQL Injection; http://www.owasp.org/index.php/Testing_for_SQL_Injection_(OWASP-DV-005)
- Web Application Attack and Audit Framework (W3AF); http://w3af.sourceforge.net/
- SAP Developer Network Secure Programming Guides; http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/334929d6-0a01-0010-45a9-8015f3951d1a

Eliminate Weak Cryptography

Over the last few years, serious weaknesses have been found in many cryptographic algorithms and their implementation, including underlying security protocols and random number generation. Due to the widespread use of cryptography for securing authentication, authorization, logging, encryption or data validation/sanitization application processes, and their confidentiality and integrity protection in particular, cryptography-related weaknesses can have a serious impact on a software application's security.

When appropriate for communication purposes, especially network communications, strong preference should be given to standardized protocols that have undergone public review—Secure Socket Layer (SSL), Transport Layer Security (TLS), IPSec, Kerberos, OASIS WS-Security, W3C XML Encryption and XML Signature, etc.—rather than using low-level cryptographic algorithms and developing a custom or unique cryptographic protocol.

If low-level cryptography must be used, only standardized cryptographic algorithms and implementations, known to be presently secure, should be used in software development. When appropriate, consideration should be given to government-approved or required algorithms. For example, U.S. 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, as well as an assessment process that provides a level of assurance of the quality of cryptographic implementations. Though vendors need to account for cryptographic export restrictions, FIPS 140-2 is an example of a sound standard to consider.

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

- MD4
- MD5
- SHA1
- Symmetric cryptographic algorithms (such as DES, which only supports 56-bit key length) imposing the use of keys shorter that 128-bits
- Stream ciphers (such as RC4 and ARC) should be discouraged due to the difficulty of using stream ciphers correctly and securely
- Block ciphers using Electronic Code Book (ECB) mode
- Any cryptographic algorithm that has not been subject to open academic peer review

The design, implementation and public review of cryptographic technology has inherent technical complexities. Even in small development projects with easy task coordination, security weaknesses can result from the improper use of cryptography. To avoid common implementation errors, applications should reuse cryptographic functions as a service, and design and implementation of proprietary cryptographic methods should be avoided. The mandatory use of the common cryptographic functions should be required by internal development standards or policies and verified as outlined below.

Application developers must use high quality random number generation functions when creating cryptographic secrets, such as encryption keys. Cryptographic code should *never* use algorithmic random number generation functions, such as rand() in C or C++, java.util.Random in Java and System.Random in C# or VB.NET.

Another key element for eliminating weak cryptography is ensuring secure management of and access to cryptographic keys. Cryptographic keys are used during program execution to perform encryption, decryption and integrity verification operations. Their exposure to malicious users via insecure program flow, configuration or mismanagement can result in serious weaknesses in the security of software applications and security protocols.

Treating keys as application data with very high security requirements and ensuring their security throughout the application lifecycle should be among the top priorities in secure application development. While at rest, keys should always be managed within a secure system configuration database, a secure file system or hardware storage location. Access to system keys must be granted explicitly to applications via key storage access control mechanisms or role assignment of the applications' users. After reading key material from a secure key, storage applications shouldn't embed or persistently store keys or key material elsewhere.

Key material must be securely erased from memory when it is no longer needed by the application.

Symmetric encryption keys are also frequently used in network communication over open networks such as the Internet. In these cases, preference should be given to asymmetric key cryptographic algorithms to distribute symmetric keys. These algorithms have, by design, lower exposure of secret key material in the remote communication, and with security protocol standardization efforts, enable more secure distribution of keys over specialized key distribution, management and revocation infrastructures.

For key protection beyond the secured endpoints, application developers should consider providing security guides to help administrators protect and manage keys used by the application.

CWE References

The CWE includes a number of cryptographic weaknesses under the following umbrella:

- CWE-310: Cryptographic Issues

Under this weakness are issues like:

- CWE-326: Inadequate Encryption Strength
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- CWE-329: Not Using a Random IV with CBC Mode
- CWE-320: Key Management Errors
- CWE-331: Insufficient Entropy
- CWE-338: Use of Cryptographically weak PRNG

Verification

Applications should be verified for compliance to internal development standards or requirements for the use of cryptographic operations.

During the application design phase, internal standards should require statements about the availability of cryptographic functions to meet the use cases and requirements outlined in application specification. Where cryptographic functions are used, the verification has to focus on driving the application planning toward prescribed guidelines for:

- The cryptography-providing libraries that should be used
- How the libraries should be accessed from within the application
- How keys should be created, accessed, used and destroyed
- Where relevant, the security protocol that should be used for exchanging cryptographic keys or communication

During application development, verification must focus on checking the source code implementation for the correct use of the prescribed guidelines and ensuring the secure handling of keys, including while they are in use or at rest. The verification can be conducted either by source code review, or by automated source code scanners. The validation should be performed in the following general directions:

- Reuse of centrally-provided cryptographic and random number functions
- Check against invocation of banned cryptographic algorithms, known to be insecure
- Check against hard-coded or self-developed functions for random number generation, encryption, integrity protection or obfuscation that shouldn't be used
- Secure management and use of keys
- Secure configuration for keys to keys by default
- Check for proper protocol selection to application interaction channels that require cryptography-based confidentiality or integrity protection

Tool or Technique	Outcome
Fortify SCA 360	None of the following warnings: C/C++: Weak Cryptographic Hash C/C++: Weak Cryptographic Hash (Hard-coded Salt) C/C++: Weak Encryption (Inadequate RSA Padding) C/C++: Weak Encryption (Insufficient Key Size) Java: Weak Cryptographic Hash (Hard-coded Salt) Java: Weak Encryption Java: Weak Encryption (Inadequate RSA Padding) Java: Weak Encryption (Insufficient Key Size) PHP: Weak Cryptographic Hash PHP: Weak Cryptographic Hash (Hard-coded Salt) PHP: Weak Encryption (Inadequate RSA Padding) PHP: Weak Encryption SQL: Weak Cryptographic Hash VB/VB.NET: Weak Cryptographic Hash VB/VB.NET: Weak Encryption (Insufficient Key Size) ColdFusion: Weak Cryptographic Hash ColdFusion: Weak Encryption JavaScript: Weak Cryptographic Hash JavaScript: Weak Encryption JavaScript: Weak Encryption (Insufficient Key Size)
Klocwork	No warnings from the "SV.FIU.POOR_ENCRYPTION" checker

Resources

References:

- NIST; Computer Security Division Computer Security Resource Center; Cryptographic Module Validation Program (CMVP); http://csrc.nist.gov/groups/STM/cmvp/index.html
- National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 140-2; Security Requirements for Cryptographic Modules; http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
- RSA Laboratories; Public-Key Cryptography Standards (PKCS); http://www.rsa.com/rsalabs/node.asp?id=2124
- Public-Key Infrastructure (X.509) (pkix);Description of Working Group; http://www.ietf.org/html.charters/pkix-charter.html
- W3C XML Encryption Work Group; http://www.w3.org/Encryption
- W3C XML Signature Work Group; http://www.w3.org/Signature
- Cryptographically secure pseudorandom number generator; http://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
- Common Criteria Portal: http://www.commoncriteriaportal.org/

Resources (continued)

Books, Articles and Reports:

- The Developer's Guide to SAP NetWeaver Security; Raepple; SAP Press; 2007.
- Cryptography Engineering: Design Principles and Practical Applications; Ferguson, Schneier and Kohno; Wiley 2010.
- The Security Development Lifecycle; Chapter 20; "SDL Minimum Cryptographic Standards"; Howard & Lipner; Microsoft Press.
- Security Engineering: A Guide to Building Dependable Distributed Systems, Chapter 5; Cryptography; Anderson; http://www.cl.cam.ac.uk/~rja14/book.html
- Programming Satan's Computer; Anderson and Needham; http://www.cl.cam.ac.uk/~rja14/Papers/satan.pdf
- SDL Crypto Code Review Macro; Howard; http://blogs.msdn.com/b/michael_howard/archive/2007/06/14/sdl-crypto-code-review-macro.aspx

Tools / Tutorials:

- Oracle ; Java SE Security Cryptography Extension; http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html
- Generic Security Services Application Program Interface; http://en.wikipedia.org/wiki/GSSAPI
  - The Generic Security Service API Version 2 update 1; http://tools.ietf.org/html/rfc2743
  - The Generic Security Service API Version 2: C-bindings; http://tools.ietf.org/html/rfc2744
- Randomness Requirements for Security; http://tools.ietf.org/html/rfc4086

Use Logging and Tracing

In the event of a security-related incident, it is important for personnel to piece together relevant details to determine what happened, and this requires secure logging. The first practice embraced by SAFECode members is to use the logging features of the operating system if possible rather than creating new logging infrastructure. Developers should use the Event Log APIs for Windows and syslog for Linux and MacOS. In some cases, it is appropriate to use non-OS logging, for example W3C log files used by web servers. The underlying infrastructure for these logging technologies is secure as they provide tamper protection. It is critically important that any logging system provide controls to prevent unauthorized tampering. Some processes, for example those running in a sandbox, may require a broker-process to hand off event data to the logging system because the process itself has insufficient rights to update log files.

Developers should log enough data to trace and correlate events, but not too much. A good example of "too much" is logging sensitive data such as passwords and credit card information. For cases where the logging of such information can't be avoided, the sensitive data has to be made hidden before it is written in the log record.

Examples of minimum information that should be logged include:

- User access authentication and authorization events
- Unambiguous username or email address
- Client machine address (IP address)
- UTC time & date
- Event code (to allow rapid filtering)
- Event description
- Event outcome (e.g. user access allowed or rejected)
- Changes to application security configuration
- Configuration changes to level of logged events
- Maintenance of log records for security or system events

A good best practice is to differentiate between monitoring logs, relevant for configuration troubleshooting, and audit logs, relevant for forensic analysis for the application security issue exploitation. This best practice helps avoid an overload of log records with useless event records. Both types of logs should be configurable during application runtime, with the configuration allowing the definition of levels of richness of logging information.

CWE References

There are three main CWE logging references software engineers should be aware of:

- CWE-778: Insufficient Logging
- CWE-779: Logging of Excessive Data
- CWE-532: Information Leak Through Log Files

Verification

Verification for the use of logging and tracing should be benchmarked to industry standards, internal development standards or the requirements of product security certification programs such as Common Criteria. In the verification process, testers should check configuration capabilities of application logging and tracing functionalities and keep in mind that the level of logging information is not standardized and is subjective to the environment in which the application operates.

The methods that can be used to verify proper use of logging and tracing include code reviews, code scans and security assessments. Results from threat modeling should also be used to evaluate the security risk exposure of the application and determine the level of necessary auditing needed.

Resources

References:

- Common Criteria for Information Technology Security Evaluation; Part 2: Security functional components; July 2009; http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R3.pdf
- IETF; RFC 5425 Transport Layer Security (TLS) Transport Mapping for Syslog; Miao, Ma and Salowey; http://tools.ietf.org/search/rfc5425

Books, Articles and Reports:

- The Security Development Lifecycle; p. 279 "Repudiation Threat Tree Pattern"; Howard & Lipner; Microsoft Press.

Tools / Tutorials:

- SAP Help Portal; Security Audit Log (BC-SEC); http://help.sap.com/saphelp_nw70ehp2/helpdata/en/68/c9d8375bc4e312e10000009b38f8cf/frameset.htm
- SAP Help Portal; Security Audit Log of AS Java; http://help.sap.com/saphelp_nw70ehp2/helpdata/en/03/37dc4c25e4344db2935f0d502af295/frameset.htm

Testing Recommendations

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 goal is not to add security by testing, but rather to validate the robustness and security of the software.

Automated testing methods are intended to find certain types of security bugs, and should be performed on the source code of all products under development because the cost of running such automated tests is low. In addition to automated tests, security test cases can be based on results from threat modeling, misuse cases (use cases that should be prevented), or previously identified bugs. Often, security test cases differ from "regular" quality assurance test cases in that instead of trying to validate expected functionality, security test cases try to uncover application failures by creating unexpected and malicious input and circumstances.

Though security testing is sometimes done as acceptance testing prior to making the product available to customers, it is likely to be more cost-effective and detect regressions and errors better when brought to an earlier phase in the software development lifecycle—to module or integration testing, for example. Security test case creation can even precede implementation, as in test or behavior-driven development models.

Determine Attack Surface

A prerequisite for effective testing is to have an up-to-date and complete understanding of the attack surface. A great deal of attack surface detail can be gathered from an up-to-date threat model. Attack surface data can also be gathered from port scanning tools and tools like Microsoft's Attack Surface Analysis Tool (see Resources).

Once the attack surface is understood, testing can then focus on areas where the risk or compliance requirements are the highest. In most cases, this includes any protocol and parser implementations that process inputs. In some cases, parts of the attack surface may be elsewhere than on the immediate external interface.

Attack surface can be determined from the product's requirements and design by looking at the inputs to the program—networking ports, IPC/RPC, user input, web interfaces, and so on, or by scanning the product, for example, with a port scanner. Periodically validating the attack surface of the actual code can also assist in preventing new vulnerabilities being opened up in the system by a change or bug fix. Products with a large attack surface or complex input processing are more susceptible to attack.

Use Appropriate Testing Tools

Different tools have different focuses. Fuzz testing tools aim to detect errors in the program code, and do not rely on knowledge of previously known

vulnerabilities, although new fuzz test cases should be added to detect any newly discovered vulnerabilities. See "Perform Fuzz/Robustness testing" below for further information about fuzz testing.

Some network and web application vulnerability scanners can also target programming errors. Some of these scanners can test against known classes of vulnerabilities such as SQL injections and cross-site scripting vulnerabilities. Many scanning tools are used by IT staff to verify their systems are correctly updated and configured rather than used by developers. But some tools, especially those that focus in finding application-level vulnerabilities, rather than administrative issues, can be very useful at finding security issues.

Network packet analyzers and network or web proxies that allow man-in-the-middle attacks and data manipulation are typically used for exploratory testing. The use of these tools often requires extensive knowledge of the underlying protocols. For example, a web proxy could be used to change session identifiers or message headers on the fly.

Automation at all stages of the testing process is important because automation can tirelessly augment human work. On the other hand, the use of automated tools will require careful setup and tweaking to get proper results. An automated tool that is blindly run against a system without understanding the system or its attack surface might not test some parts of the system at all, or test it with the wrong type of inputs. The risk of this happening is typically larger if test tools are run by an external group that may not have complete understanding on the system.

Perform Fuzz / Robustness 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 maturing rapidly. Fuzz testing tools for standard protocols and general use are available, but in some cases software developers must build bespoke fuzz testers to suit specialized file and network data formats used by their application. Fuzz testing is an effective testing technique because it uncovers weaknesses in data-handling code that may have been missed by code reviews or static analysis.

The process of fuzz testing can be lengthy, so automation is critical. It is also important that priority be given to higher exposure entry points for fuzz testing, for example, an unauthenticated and remotely accessible TCP port, because higher exposure entry points are more accessible to attackers.

In order to perform effective fuzz testing, select tools that best support the networking protocols or data formats in use. If none can be found in the marketplace, fuzz test tools should be built. Though the low-level process required to build effective fuzz tools is beyond the scope of this paper, the Resources section below provides some references for readers interested in learning more.

Fuzz testing is not static. Fuzz testing cases should evolve as new vulnerabilities are found. For example, if a vulnerability is discovered in the application's file parser, a fuzz test case should be created that would trigger that condition. This new test case should be added to the library of tests that are run regularly against the application. In some cases, a new fuzzer may be needed if the data format has not been previously fuzz tested.

Fuzz testing may be used in conjunction with other testing types. For example, a more focused vulnerability scanner can be used to inject fuzz inputs to the target product.

Perform Penetration Testing

The goal of penetration testing is to break the system by applying testing techniques usually employed by attackers, either manually or by using attack tools. Penetration testing is a valuable tool for discovering vulnerabilities that reside in the system's business logic. High-level business logic aspects are often hard to detect from the code level. However, it is important to realize that a penetration test cannot make up for an insecure design or poor development and testing practices.

Some SAFECode members have dedicated penetration testing teams while others employ external penetration and security assessment vendors. Some SAFECode members use both in-house and external security penetration expertise. Penetration testing should be performed along with standard functional testing as part of a comprehensive test plan.

Penetration test cases can be based on "misuse cases" or "attacker stories," requirements that specify what should not be possible.

The advantage of using competent, third-party penetration testers is their breadth of experience. The challenge is finding third-party testers that will do an effective job for the 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.

It should be stressed that testing is not a replacement for a development process that helps build more secure software, but rather that security testing is a core part of such a software development process.

CWE References

Security testing should cover any aspect of the system or application and therefore should validate the effectiveness of controls for all types of weaknesses.

Fuzz testing mainly targets exception and incorrect input handling (CWE-20). However, sometimes the input might be valid, but mishandled by the application.

First-line input handling weaknesses include, for example:

- CWE-118: Improper Access of Indexable Resource
- CWE-703: Failure to Handle Exceptional Conditions
- CWE-228: Improper Handling of Syntactically Invalid Structure
- CWE-237: Improper Handling of Structural Elements
- CWE-229: Improper Handling of Values
- CWE-233: Parameter Problems

Protocol-level security testing is useful for detecting, for example, weaknesses related to CWE-693: Protection Mechanism Failure, such as CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade') or CWE-345: Insufficient Verification of Data Authenticity.

Penetration testing could, in theory, find any type of weakness depending on the skill of the people performing the penetration test.

Verification

The existence of security testing can be verified by evaluating:

- Documented business risks or compliance requirements that provide prioritization for all testing activities. Failed or missed test cases should be evaluated against these.
- Mitigating controls to identified threats, abuse cases, or attacker stories as requirements
- Security test case descriptions
- Security test results
- Penetration testing or security assessment reports

Resources

Attack surface tools include:

- Process Explorer: http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx
- WinObj: http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx
- Determining open ports can be done, for example, using nmap (http://nmap.org/)
- On Unix systems, listing open files can be done with the lsof command, open ports can be viewed with netstat, and running processes and which files they are opening can be traced with strace.
- Attack Surface Analyzer - Beta http://www.microsoft.com/downloads/en/details.aspx?FamilyID=1283b765-f57d-4ebb-8f0a-c49c746b44b9

Resources (continued)

Examples of software security testing references include:

- The Art of Software Security Testing: Identifying Software Security Flaws; Wysopal, Nelson, Dai Zovi & Dustin; Addison-Wesley 2006.
- Open Source Security Testing Methodology Manual. ISECOM, http://www.isecom.org/
- Common Attack Pattern Enumeration and Classification. MITRE, http://capec.mitre.org/

Examples of common fuzz testers are listed below. Different test tools are useful for different targets, and sometimes it is necessary to build an additional tool to actually get the malformed data to the right place (for example, fuzzing a compressed file tests the compression layer but not necessarily the parser for the data that had been compressed).

- Zzuf: http://caca.zoy.org/wiki/zzuf
- Peach: http://peachfuzzer.com/
- Radamsa: https://code.google.com/p/ouspg/wiki/Radamsa
- Untidy: http://untidy.sourceforge.net/
- MiniFuzz: http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=b2307ca4-638f-4641-9946-dc0a5abe8513
- SDL Regex Fuzzer; http://www.microsoft.com/downloads/en/details.aspx?FamilyID=8737519c-52d3-4291-9034-caa71855451f

Examples of protocol testing and proxy tools include:

- Scapy: http://www.secdev.org/projects/scapy
- PortSwigger Web Security; Burp Proxy; http://portswigger.net/burp/proxy.html

Other fuzz testing resources include:

- Fuzzing: Brute Force Vulnerability Discovery; Sutton, Greene, & Amini, Addison-Wesley
- Fuzzing Reader - Lessons Learned; Randolph; December 1, 2009 http://blogs.adobe.com/asset/2009/12/fuzzing_reader_-_lessons_learned.html
- BlueHat v8: Fuzzed Enough? When it's OK to Put the Shears Down; http://technet.microsoft.com/en-us/security/dd285263.aspx
- Writing Fuzzable Code; Microsoft Security Development Lifecycle; http://blogs.msdn.com/b/sdl/archive/2010/07/07/writing-fuzzable-code.aspx

Technology Recommendations

Use a Current Compiler Toolset

As noted earlier in this paper, memory-corruption issues, including buffer overruns and underruns, are a common source of vulnerabilities in C and C++ code. It is easy to fix many memory-corruption issues by moving away from low-level languages like C and C++ to higher-level languages such as Java or C# for new projects. However, using a new programming language is much harder to do in practice because the migration cost of training and hiring can be expensive, and time-to-market can be put at risk as engineers grapple with the nuances inherent in an updated toolset. There is also a very large base of legacy C and C++ code in the marketplace that must be maintained. Finally, for some classes of software, C or C++ is the most appropriate programming language, and the languages are ubiquitous. Because memory-corruption 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 memory-corruption bugs automatically. Such defenses can make it harder for exploit code to execute predictably and correctly. Examples of defenses common in C and C++ compilers include:

- Stack-based buffer overrun detection
- Address space layout randomization
- Non-executable memory
- Insecure code warnings
- Safe exception handling
- Automatic migration of insecure code to secure code

The two most common C and C++ compilers are Microsoft Visual C++ and GNU's gcc. Because of the security enhancements in newer versions of each of these tools, software development organizations should use:

- Microsoft Visual C++ 2008 SP1 or later. Microsoft Visual C++ 2010 is preferred owing to better stack-based buffer overrun defenses.
- gcc 4.4.x or later.

Software development organizations should compile and/or link native C and C++ code with the following options:

- Microsoft Visual C++
  - /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
  - /we4996 for insecure C runtime function detection and removal (see "Minimize unsafe function use")

- gcc
  - -fstack-protector or -fstack-protector-all for stack-based buffer overrun defenses
  - -fpie -pie for image randomization
  - -D_FORTIFY_SOURCE=2 and -Wformat-security for insecure C runtime function detection and removal (see "Minimize use of unsafe functions")
  - -ftrapv to detect some classes of integer arithmetic issues (see "Audit dynamic memory allocations and array offsets")

While this topic mainly focuses on native C and C++ code, other toolsets can take advantage of operating system defenses, such as address space layout randomization and non-executable memory. Examples include:

- Microsoft Visual C# 2008 SP1 and later (address space layout randomization and non-executable data memory by default)
- Microsoft Visual Basic 2008 SP1 and later (address space layout randomization and non-executable data memory by default)

CWE References

Most of the defenses added by the compiler or linker address memory-corruption issues such as:

- CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-805: Buffer Access with Incorrect Length Value
- CWE-129: Improper Validation of Array Index
- CWE-190: Integer Overflow or Wraparound
- CWE-131: Incorrect Calculation of Buffer Size

Verification

A Microsoft tool named the BinScope Binary Analyzer can verify if most of the compiler and linker options (/GS, /DYNAMICBASE, /NXCOMPAT and /SAFESEH) are enabled in a Windows image. The tool should yield a "Pass" for every binary that ships with an application.

Verifying that /we4996 is enabled requires looking for the compiler setting in all build files, or looking for the following line of code in an application-wide header file:

```
#pragma warning(3 : 4996)
```

Developers can verify that gcc-compiled applications are position independent with the following command-line instruction:

```
readelf -h <filename> | grep Type
```

Position independent executables are type "DYN"

Resources

References:

- Hardened Linux from Scratch – Version SVN-20080603; Chapter 2.6 Position Independent Executables; http://linuxfromscratch.xtra-net.org/hlfs/view/unstable/glibc-2.4/chapter02/pie.html

Books, Articles, and Reports

- MSDN Library; Windows ISV Software Security Defenses; Howard, Miller, Lambert & Thomlinson; December 2010; http://msdn.microsoft.com/en-us/library/bb430720.aspx

Presentations:

- Exploit Mitigation Techniques (in OpenBSD, of course); The OpenBSD Project; de Raadat; http://www.openbsd.org/papers/ven05-deraadt/index.html

Tools / Tutorials :

- BinScope Binary Analyzer: http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=90e6181c-5905-4799-826a-772eafd4440a
- Patch: Object size checking to prevent (some) buffer overflows: http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html
- GCC extension for protecting applications from stack-smashing attacks: http://www.trl.ibm.com/projects/security/ssp/
- Process Explorer: http://technet.microsoft.com/en-us/sysinternals/bb896653

Use Static Analysis Tools

Static analysis tools are now commonly used by development organizations, and the use of such tools is highly recommended to find common vulnerability types.

Static code analysis tools can help to ensure coding mistakes are caught and corrected as soon as possible. Tools that integrate with development environments are usually considered easier to use and often lead to faster bug resolution; they also help get developers used to identifying security defects as they develop code and before they check-in. Using static analysis tools that are integrated with development environments does not replace the need for codebase-wide analysis. Developers may have a modified view of the current code base (e.g., on a dedicated maintenance branch) or may only be dealing with a limited set of source code (e.g., one module or application tier). Both scenarios can result in false negatives resulting from limited data flow and control flow analysis and other problems that full-codebase and/or main branch analysis (at product build time) would otherwise find.

Ideally, static code analysis tools should be site licensed to the entire development team, including QA, making this tool as commonly used by the development team as spell checkers that are built in to modern word processors. Both experienced and inexperience developers can greatly benefit from analysis tools much like all writers take advantage of spell checkers. Because many vulnerabilities are hard to spot but simple to solve, it's not unreasonable to expect most vulnerabilities to be fixed immediately after a routine scan completes. Performing a Threat Model before starting a code analysis effort can also help in the triage process, as it can help focus auditors on critical or risky components, getting defects from those areas prioritized to be addressed first.

First time static analysis tools users should expect some up-front investment to get the greatest benefit from the tools. Before running a static analysis tool for the first time, it is recommended to clean the code from compiling warnings. Still, an initial run will result in a significant list of findings. Depending on the project size, management should consider dedicating team resources to do the initial triage. Once triage is complete, some findings may be determined to be false due to contextual information the static analysis tool does not have, and some issues that were considered by the tool to be less severe may be elevated in priority to be addressed (again due to context, such as business risk or other factors, which the tool is not aware). Tuning the tool and the code using standard annotation language (SAL) will often result in fewer false findings, and providing training to developers can greatly aid in the triage effort as they become more familiar both with the tool output and software security concepts. Maintaining a dedicated team of security-savvy developers to review static analysis results may be helpful for resource-constrained

development teams, but in the long run does the team a disservice by masking or hiding results, both good and bad, from the folks who created them. Once a tree is clean of static analysis warnings, the revision control system should be configured to prohibit check-ins of code that introduces new warnings and the code needs to be regularly audited for pragmas that disable warnings. Development teams often create a separate build system with static analysis tools running continuously. This practice minimizes the impact on the time it takes to generate a new build.

Several static code analysis tools are capable of generating results even if the codebase is incomplete or does not compile. While teams may greatly benefit from testing code before reaching integration checkpoints, analyzing code that does not compile is highly discouraged as it yields suboptimal results. It's also important to understand that static code analysis tools are a complement to manual code review, not a substitute. A clean run does not guarantee the code is perfect. It merely indicates the code is free of well-known and well-understood patterns.

Static analysis tools really shine when a new vulnerability is discovered: automated tools can perform an initial assessment of a large body of software a lot quicker than manual code review can be performed. Many static analysis tools operate using flexible and extensible rules, which can be added to when new vulnerability classes are discovered or modified for changes in common APIs. New rules can often be added to account for internal coding standards or APIs (e.g., to indicate certain internally-developed interfaces affect the security of code passing through them, either negatively or positively). Caution must be taken when updating rules between builds, especially in large complex codebases—modifying existing rules (for analysis bugs discovered) may result in a reduction of findings as analysis improves, but adding new rules for new issues may result in additional findings. These new findings would need to be triaged and may result in spikes in metrics not due to anything done by developers (i.e. adding new code). Rule updates should be planned to keep up-to-date with changes in the security landscape without throwing a project off its rails.

Depending on the codebase size, a full analysis can take a considerable amount of time to run. Tuning can help reduce the time required for analysis. It is also recommended to reduce the initial set of things that the tool looks for, such as to specific security issues, or simply to security issues only (rather than traditional quality defects, like memory leaks, which are better discovered by other tools). This initial modification to what is being analyzed can help reduce analysis time and may result in fewer findings leading to better overall adoption. Then, as development teams get more comfortable with the tool, they can open up the rule set to find more issues. Some tools also perform analysis in two or more stages, usually a build stage and a separate analysis stage. The analysis stage can be

performed in parallel with other build activities (such as linking or dynamic testing) and can take advantage of dedicated processing power and CPU/disk resources, which can speed up analysis.

Regardless of the tool and the type of technology employed, no one tool today finds all faults. In fact, all SAFECode companies employ multiple tools throughout the development lifecycle. Furthermore, neither static nor dynamic analysis can recognize sophisticated attack patterns or business logic flaws, so they should not be considered a replacement for code reviews. While tools can reliably identify vulnerability types, automated severity metrics cannot be taken for granted as they don't factor business risk such as asset value, cost of down time, potential for law suits and impact of brand reputation.

CWE References

Static analysis tools find a plethora of security vulnerabilities, so one could argue that many CWEs can be found through the use of analysis tools.

Verification

Static analysis tools are themselves a form of verification. While a clean analysis tool run does not imply an application is secure, it is a good indicator of rigor by the development team.

Resources

References:

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

Books, Articles, and Reports:

- Secure Programming with Static Analysis; Chess & West; Addison-Wesley 2007.
- The Security Development Lifecycle; Chapter 21 "SDL-Required Tools and Compiler Options"; Howard & Lipner; Microsoft Press.
- SecurityInnovation; Hacker Report: Static Analysis Tools, November 2004 Edition; http://www.securityinnovation.com/pdf/si-report-static-analysis.pdf
- Cigital Justice League Blog; Badness-ometers are good. Do you own one?; McGraw; http://www.cigital.com/justiceleague/2007/03/19/badness-ometers-are-good-do-you-own-one/

Presentations:

- Using Static Analysis for Software Defect Detection; William Pugh; July 6, 2006; http://video.google.com/videoplay?docid=-8150751070230264609

Tools / Tutorials:

- MSDN Library; Analyzing C/C++ Code Quality by Using Code Analysis; http://msdn.microsoft.com/en-us/library/ms182025.aspx
- MSDN Library; FxCop; http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx

Summary of Practices

Section	Practice	Page number
Secure Design Principles	Threat Modeling	2
	Use Least Privilege	7
	Implement Sandboxing	10
Secure Coding Practices	Minimize Use of Unsafe String and Buffer Functions	12
	Validate Input and Output to Mitigate Common Vulnerabilities	15
	Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets	19
	Use Anti-Cross Site Scripting (XSS) Libraries	22
	Use Canonical Data Formats	27
	Avoid String Concatenation for Dynamic SQL Statements	29
	Eliminate Weak Cryptography	32
	Use Logging and Tracing	37
Testing Recommendations	Determine Attack Surface	39
	Use Appropriate Testing Tools	39
	Perform Fuzz / Robustness Testing	40
	Perform Penetration Testing	41
Technology Recommendations	Use a Current Compiler Toolset	44
	Use Static Analysis Tools	47

Moving Industry Forward

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

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

Acknowledgements

Brad Arkin, Adobe Systems Incorporated

Eric Baize, EMC Corporation

Gunter Bitz, SAP AG

Danny Dhillon, EMC Corporation

Robert Dix, Juniper Networks

Steve Lipner, Microsoft Corp.

Gary Phillips, Symantec Corp.

Alexandr Seleznyov, Nokia

Janne Uusilehto, Nokia

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.

Product and service names mentioned herein are the trademarks of their respective owners.

SAFECode
2101 Wilson Boulevard
Suite 1000
Arlington, VA 22201

(p) 703.812.9199
(f) 703.812.9350
(email) stacy@safecode.org
www.safecode.org

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