CS 465

24 DEADLY SINS

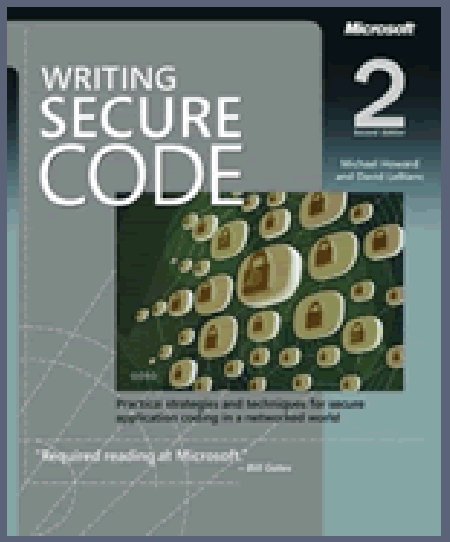
Slides by Kent Seamons and Tim van der Horst
Last Updated: Nov 11, 2011

Software Security

Lots to Learn

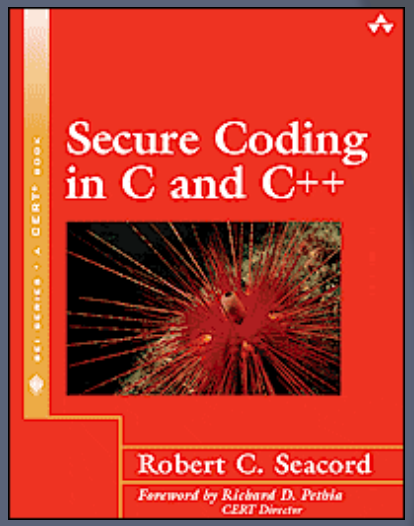
- As a software professional, you need to educate yourself on how to build secure software
- We only cover the tip of the iceberg
- You should read several books on the topic
  - Make it a habit to stay current with the latest thinking
  - Microsoft security lockdown

Recommended Books



Writing Secure Code, 2nd Edition
Howard and LeBlanc

Building Secure Software
Viega and McGraw



Secure Coding in C and C++
Robert Seacord

Genesis of the ~~19~~ 24 Deadly Sins

- In early 2004, Amit Yoran (Director of National Cyber Security Division, Department of Homeland Security) announced that 95% of software security bugs arise from 19 programming mistakes
- 2010 revised and updated
- Objectives: short, highly actionable, to the point
- Learn to avoid these mistakes!

Who Should Read This Book?

- Designer
- Coder
- Tester

What Should You Read?

- Web Application Sins
  - Build web applications (client of server)
- Implementation Sins
  - Language-specific implementation issues
- Cryptographic Sins
  - Application performs cryptography
- Networking Sins
  - Application performs network communication
- All Developers – 10,11,12,and 14
- Developers of applications that require frequent updating – 15
- Developers of languages that support exceptions – 9
- Develops of C, C++ - 6, 7, and 8
- Just in time training – review relevant items before development

Web Application Sins

Sin 1 – SQL Injection

- Typically the result of an attacker providing mal-formed data to an application that uses it to construct an SQL command
  - `exec (@query)`
  - User inputs an ID and the system constructs a command
    `SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''`
  - The attacker can add extra data and a comment character
    `1 or 2>1 –`
  - The result is that the query returns the entire customer table

Sin 1 – SQL Injection

- Do understand the database you use (stored procedures, comment character, etc.)
- Do understand common SQL injection attack methods against the database you use
- Do check for input validity at the server
- Do use parameterized queries
- Do use quoting or delimiting functions if you categorically must build dynamic SQL
- Do store the database connection information in a location outside of the application
- Do encrypt sensitive data
- Do deny access to underlying database objects and grant access only to stored procedures and views

Sin 1 – SQL Injection

- Do not simply strip out bad words: imagine removing “delete” from “deldeleteete”
- Do not trust input used to build SQL statements
- Do not use string concatenation to build SQL statements
- Do not execute untrusted parameters within stored procedures
- Do not check for input validity only at the client
- Do not connect to the database as a highly privileged account
- Do not embed the database login password in the application or connection string
- Do not store the database configuration information in the web root
- Consider removing access to all user-defined tables in the database, and granting access only through stored procedures. Then build the query string using stored procedure and parameterized queries.

Sin 2 – Web Server Vulnerabilities

- Do check all web-based input for validity and trustworthiness
- Do encode all output originating from user input
- Do mark cookies as HttpOnly
- Do add timestamps or timeouts to sessions that are subject to XSRF attacks
- Do regularly test your Web application's entry points with malformed data escaped script input to test for XSS and related vulnerabilities
- Do stay on top of new XSS-style vulnerabilities, as it's a constantly evolving minefield

Sin 2 – Web Server Vulnerabilities

- Do not echo web-based input without checking for validity first
- Do not rely on “disallowed” lists (blacklists) as a sole defense
- Do not change server state with GET requests
- Do not store sensitive data in cookies
- Do not expect TLS to help prevent any of these sins
- Do not use GET requests for operations that change server data
- Consider using as many extra defenses as possible

Sin 4 – Use of Magic URLS and Hidden Form Fields

- An application encodes authentication information in a URL and sends it in the clear
- The server stores information in a hidden field and assumes the user cannot see it or tamper with it
  - Some web sites have included the price in a hidden field and used that value to process a transaction

Sin 4 – Use of Magic URLS and Hidden Form Fields

- Do test all web input with malicious input
- Do not embed confidential data in any HTTP or HTML construct if the channel is not encrypted
- Do not trust any data in a web form
- Do not think the application is safe just because you use cryptography

Implementation Sins

Sin 5 – Buffer Overruns

- Do carefully check your buffer accesses by using safe string and buffer handling functions
- Do understand the implications of any custom buffer-copying code you have written
- Use compiler-based defenses such as /GS and ProPolice
- Do use operating system-level buffer overrun defenses such as DEP and PAX
- Do use address randomization where possible such as ASLR in Windows
- Do understand what data the attacker controls, and manage that data safely in your code

Sin 5 – Buffer Overruns

- Do NOT think that compiler and OS defenses are sufficient – They aren't!
  - They are simply extra defenses
- Do not create new code that uses unsafe functions
- Consider updating your C/C++ compiler, since the compiler authors add more defenses to the generated code
- Consider removing unsafe functions from old code over time
- Consider using C++ string and container classes rather than low-level C string functions

Sin 6 – Format String Problems

- What is it? A recent issue discussed publicly since only 2000
  - In C/C++, a format string bug can allow an attacker to write to arbitrary memory locations
  - Examples
    - printf (user_input_without_validation)
    - fprintf(STDOUT, user_input)
  - When a user inputs "%x %x", the output is data on the stack
  - The %n designation writes the number of characters written so far to the address of the variable in the corresponding argument
- http://www.sans.org/resources/malwarefaq/LPRng.php

Sin 6 – Format String Problems

- Do use fixed format strings, or format strings from a trusted source
- Do check and limit locale requests to valid values
- Do heed the warnings and errors from your compiler
- Do not pass user input directly as the format string to formatting functions
- Consider using higher-level languages that tend to be less vulnerable to this issue

Sin 7 – Integer Overflows

- Integer overflow and underflow, and arithmetic overflows of all types can cause crashes, logic errors, escalation of privileges, and execution of arbitrary code
- Required reading on the lecture page
  - http://msdn2.microsoft.com/en-us/library/ms972818.aspx

Sin 7 – Integer Overflows

- Do check all calculations used to determine memory allocations to check the arithmetic cannot overflow
- Do check all calculations used to determine array indexes to check the arithmetic cannot overflow
- Do use unsigned integers for array offsets and memory allocations sizes
- Do check for truncation and sign issues when taking differences of pointers, and working with size_t
- Do not think languages other than C/C++ are immune to integer overflows

Sin 8 – C++ Catastrophes

- Do use STL containers instead of manually created arrays
- Do write copy constructors and assignment operators, or declare them private with no implementation
- Do initialize all of your variables-better yet, use classes that ensure initialization always happens
- Do not mix up array new and delete with ordinary new and delete
- Do not write complex constructors that leave objects in an indeterminate state if the constructor does not complete. Better yet, write constructors that cannot throw exceptions or fail.
- Consider resetting class members-especially pointers-to a known safe state in the destructor

Sin 9 – Catching Exceptions

- Do catch only specific exceptions.
- Do handle only structured exceptions that your code can handle.
- Do handle signals with safe functions.
- Do not catch(….)
- Do not catch(Exception)
- Do not __except(EXCEPTION_EXECUTE_HANDLER)
- Do not handle SIG_SEGV signals, except to log

Sin 10 – Command Injection

- Command injection problems occur when untrusted data is passed to a compiler or interpreter that might execute the data if it is formatted in a certain way
- Example
  - How to Remove Meta-characters From User-Supplied Data in CGI Scripts
    http://www.cert.org/tech_tips/cgi_metacharacters.html
  - Taint mode in Ruby and Perl

    Locking Ruby in the Safe
    http://phrogz.net/programmingruby/taint.html

Sin 10 – Command Injection

- Redemptive Steps
  - Check the data to make sure it is ok
  - Take an appropriate action when the data is invalid
  - Run your application using least privilege. It usually isn't very amusing to run arbitrary commands as "nobody" or guest.
- Data Validation
  - Deny-list approach
  - Allow-list approach
  - "Quoting" approach

Sin 10 – Command Injection

- Do perform input validation on all input before passing it to a command processor
- Do handle the failure securely if an input validation check fails
- Do use taint defenses if your environment supports it
- Do not pass unvalidated input to any command processor, even if the intent is that the input will just be data
- Do not use the deny list approach, unless you are 100 percent sure you are accounting for all possibilities
- Consider avoiding regular expressions for user input validations; instead write a simple and clear validators by hand