Software Security

By
Hunter Stevenson
Khalid Alharbi

What is the talk NOT about?

- Cryptography.
- Database security.
- Operating Systems security.
- Network security.
- Security software.
- Encryption, digital signatures, and authentication protocols.

What is the talk about?

- **Part 1:**
  - Fundamentals of software security.
  - Software security design principles.
  - Building secure software systems.
- **Part 2:**
  - Common software vulnerabilities.

Part I:

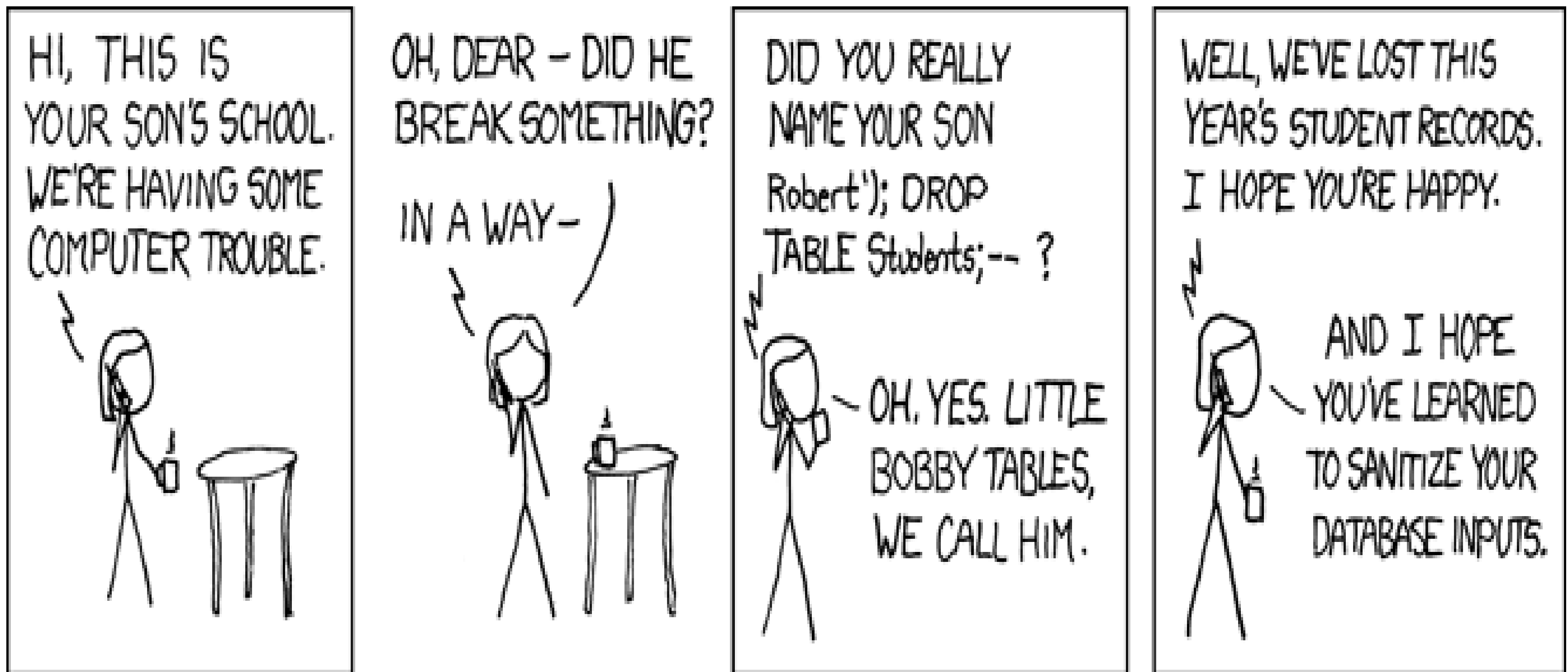
Software Security Fundamentals and Best Practices

What is Software Security?

- Software security is the idea of engineering software so that it continues to function correctly under malicious attack.
- Software Security aims to avoid security vulnerabilities by addressing security from the early stages of software development life cycle.
- "Security is a risk management."

Why Software Security?

SQL Injection!!



comic: http://xkcd.com/327/

Why Software Security?

- Most software systems today contain numerous flaws and bugs that get exploited by attackers.
- New threats emerge everyday.
- Convenience trumps security measures.
- Exponential increase in vulnerabilities in software systems.
- Software security is everybody's job.
- Programmers have a long history of repeating the same security-related mistakes!

Recent Stories (I)

- 2012 - A security flaw in *Google Wallet* that leads into full access to your Google Wallet account without extra app or rooting.
  - Your Google Wallet account is tied to the device itself but not to the account.
- 2011 - Oracle's MySQL.com hacked via SQL Injection Attack!!
- 2011 - Expedia's TripAdvisor member data stolen in possible SQL Injection Attack.
- 2010 - Hacker gained access to the Royal Navy website using SQL injection attack.

Recent Stories (II)

- 2009 - A security flaw in the *Spotify* service, by which private account information were exposed.
- 2006 - Hacker gained access to 800,000 UCLA students, faculty, and staff data.
- 2005 - A buffer overflow was deducted in *Symantec pcAnywhere* that could lead into a denial of service attack.
- 2005 - University of Southern California's online system applications were vulnerable to SQL Injection.

Terminology (I)

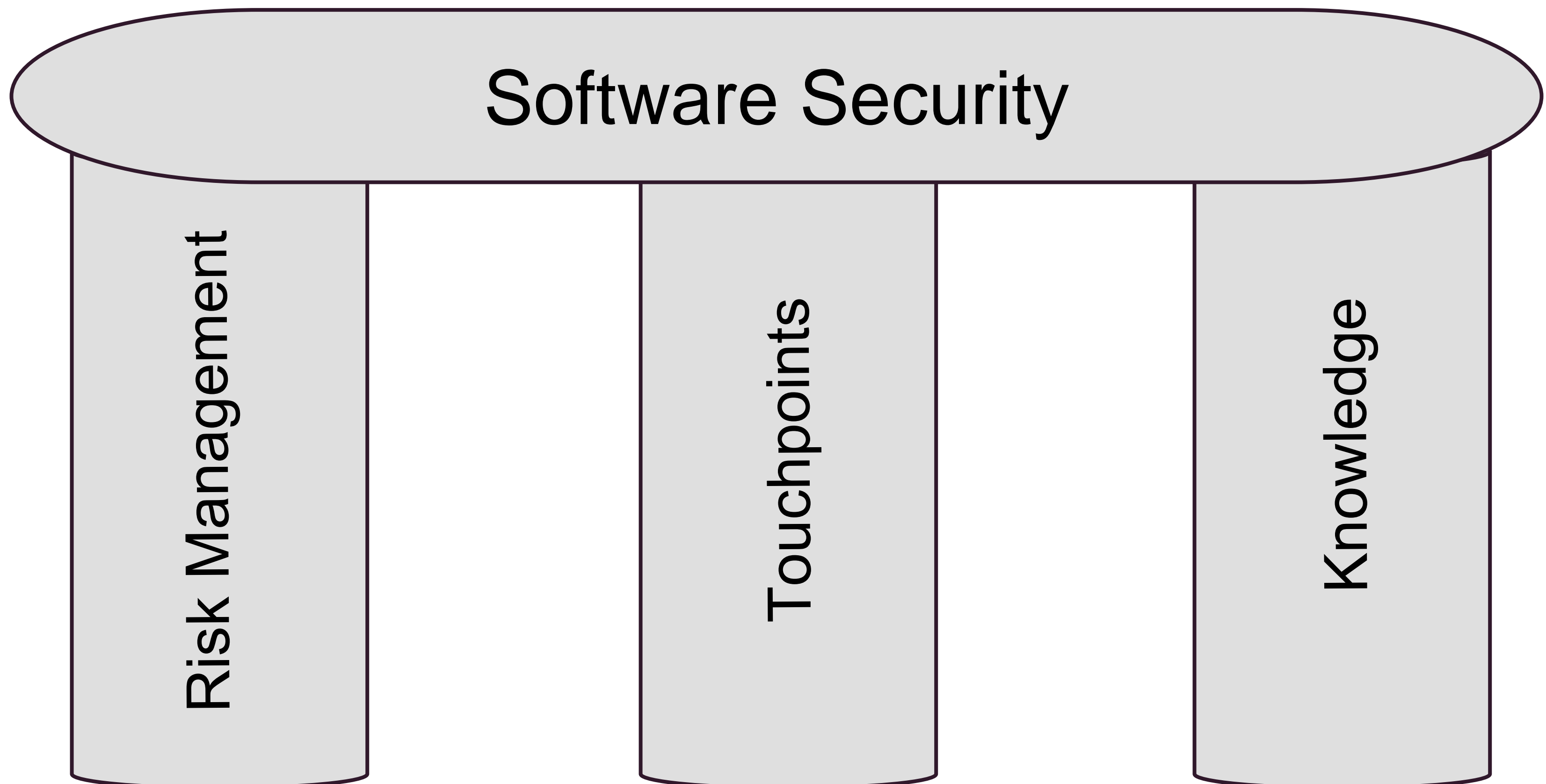
- **Defects** are implementation vulnerabilities and design vulnerabilities.
- **Bugs** are implementation-level errors that can be detected and removed.
  - Example: Buffer overflow.
- **Flaws** are problems at a deeper level. They are instantiated in the code and present or absent at design-level.
  - Example: Error-handling problems.
- **Failures** are the inability of the software to perform its required function.

Terminology (II)

- **Risks** capture the probability that a flaw or a bug will impact the purpose of the software.
  - Risk = probability x impact
- **Vulnerabilities** are errors that an attacker can exploit.
  - Either flaws in the design or flaws in the implementation.
  - Design-level vulnerabilities are the hardest defects to handle.

Pillars of Software Security

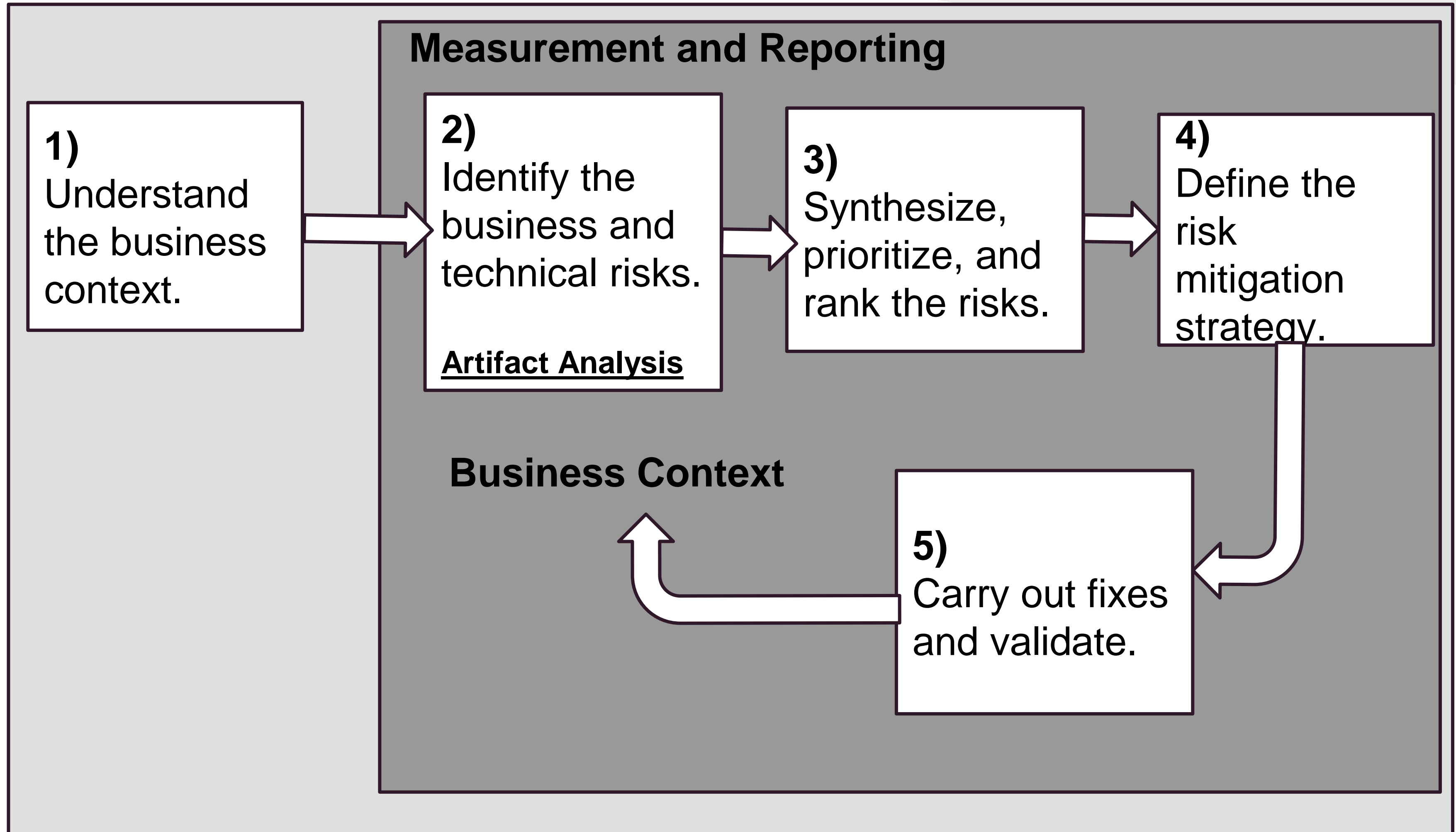
- The three pillars of software security



Pillar I: Risk Management

- A continuous risk management process is an essential part to software security.
- It identifies, ranks, tracks, and understands software security risks.
- **Risk management framework (RMF)**
  - An overall approach to risk management.
  - Allows a consistent and continuous expertise-driven approach to risk management.
  - The goal is to consistently track and handle risks.

RMF Activities (I)



RMF Activities (II)

1- Understand the business context.

- A key task of an analyst.
- Extract and describe business goals.
- Set priorities.
- Understanding what risks to consider.
- Gathering the artifacts.
- Conducting project research to the scope.

RMF Activities (III)

2- Identify the business and technical risks, synthesize, prioritize, and rank the risks.

- Business risks impact business goals.
- Mapping technical risks to business goals.
- Developing a set of risk questionnaires.
- Interviewing the target project team.
- Analyzing the research interview data.
- Evaluating software artifacts.

RMF Activities (IV)

3- Synthesize, prioritize, and rank the risks.

- Prioritize the risks based on the business goals.
- Riskmetrics:
  - Risk likelihood.
  - Risk impact.
  - Number of risks emerging over time.
- What shall we do first given the current risk situation?
- What is the best allocation of resources?

RMF Activities (V)

4- Define the risk mitigation strategy.

- Create a coherent strategy for mitigating the risks that takes into account:
  - Cost.
  - Implementation time.
  - Likelihood of success.
  - Competence.
  - Impact.
- Identify the validation techniques.
- Metrics are financial in nature.

RMF Activities (VI)

5- Carry out fixes and validate that they are correct.

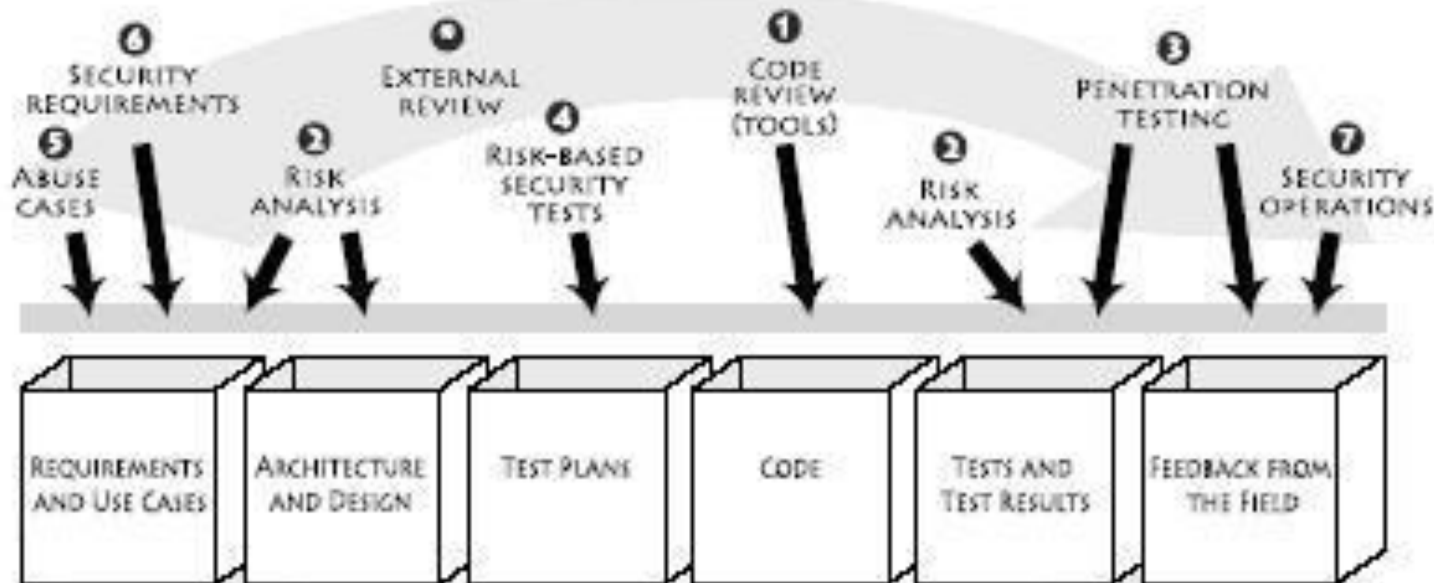
- Implement the mitigation strategy.
- The artifacts should be rectified.
- Progress is measured in terms of completeness against mitigation strategy.
- Use validation techniques to validate that artifacts no longer bear unacceptable risk.
- Metrics include artifact quality metrics and levels of risk mitigation effectiveness.

RMF Activities (VII)

- Risk management is a central software security practice.
- Successful use of RMF relies on continuous and consistent identification of risks.
- Use project management tools to track risk information.
  - Example: Open Workbench.
- RMF is a multilevel loop.
  - Identifying risks only once during the project is incorrect.
  - The five fundamental activities need to be applied repeatedly throughout the project.

Pillar II: Touchpoints

- Security touchpoints are set of security best practices.



Seven Touchpoints (I)

1. **Code Reviews.**
   - Artifact: Code.
   - Example of risks found: Buffer overflow on line 30.
2. **Architectural Risk Analysis.**
   - Artifact: Design and specifications.
   - Example of risks found: Failure of a Web Service to authenticate calling code.
3. **Penetration Testing.**
   - Artifact: System in its environment.
   - Example of risks found: Poor handling of program state in Web interface.

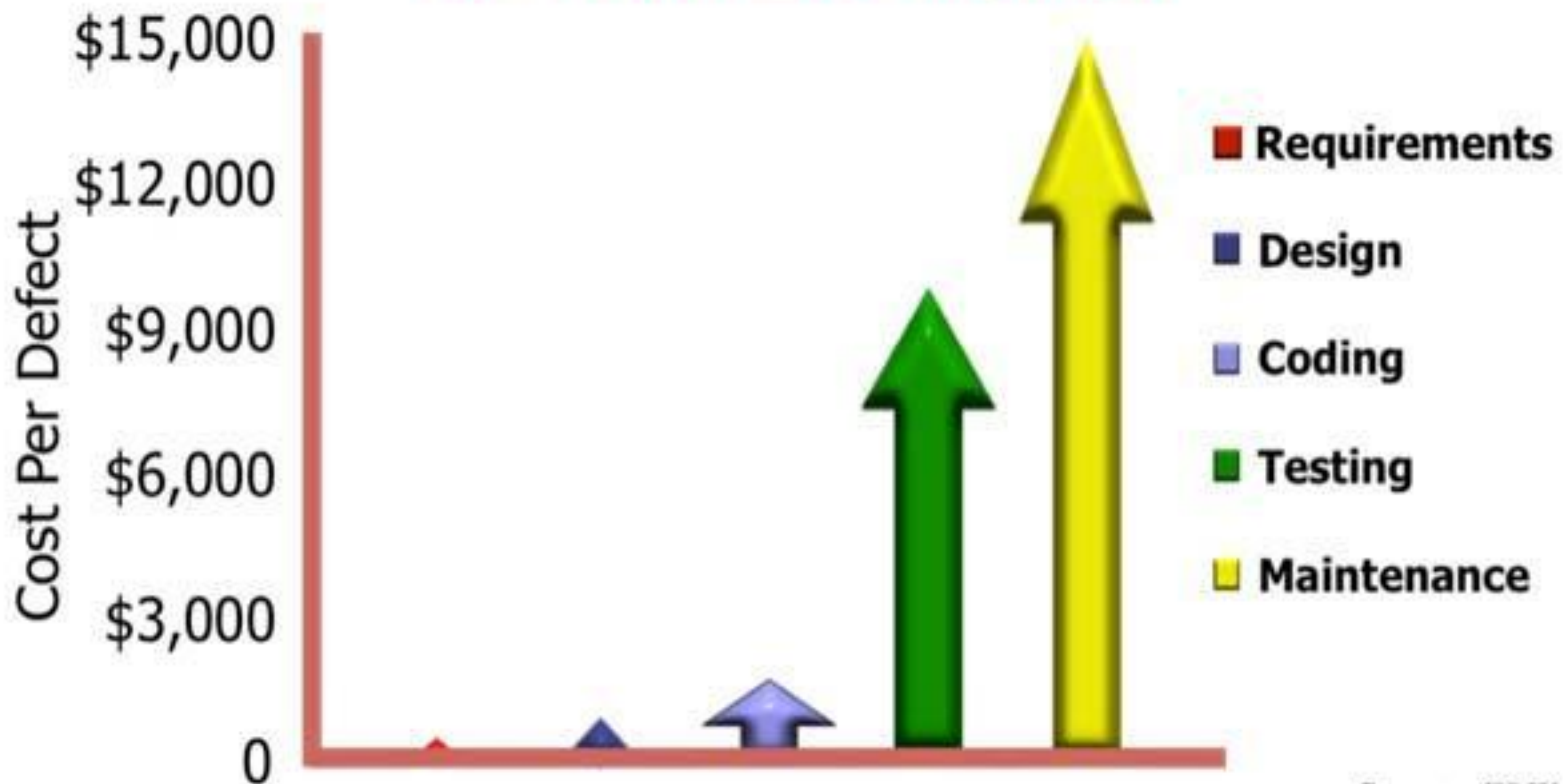
Seven Touchpoints (II)

4. **Risk-Based Security Testing.**
   - Artifact: Units and system.
   - Example of risks found: Extent of data leakage possible by leveraging data protection risk.
5. **Abuse cases.**
   - Artifact: Requirements and use cases.
   - Example of risks found: Susceptibility to well-known tampering attack.

Seven Touchpoints (III)

6. **Security Requirements.**
    - Artifact: Requirements.
    - Example of risks found: Explicit description of data protection needs is missing.
7. **Security Operations.**
    - Artifact: Fielded system.
    - Example of risks found: Insufficient logging to prosecute a known attacker.

Fixing defects: Early is better!



1- Code Review (I)

- Source code analysis.
- Focus is on finding and fixing bugs.
- Use static analysis tools to find security bugs.
  - Static analysis tools suffer from *false negatives* and *false positives*.
  - Examples: Coverity, Fortify, Ounce Labs, and Secure Software.

1- Code Review (II)

- Useful source code analysis tools must:

  1- Be designed for security.

  2- Support multiple tires.

  3- Be extensible.

  4- Be useful for security analysts and developers alike.

  5- Support existing development processes.

  6- Make sense to multiple stakeholders.

2- Architectural Risk Analysis (I)

- Design flaws:
  - Account for 50% of security problems.
  - Can't be detected by staring at code.
  - A higher-level understanding is required.
- Architectural Risk Analysis:
  - Tracks risk over time.
  - Links system-level concerns to probability and impact measures.
  - Fits with RMF.

2- Architectural Risk Analysis (II)

- Critical Aspects of Architectural Risk Analysis:
  - Attack Resistance Analysis.
  - Ambiguity Analysis.
  - Weakness Analysis.

2- Architectural Risk Analysis (III)

- **Attack Resistance Analysis:**
  - It captures the check-list like approach to the risk analysis taken in Microsoft STRIDE approach.
  - Steps:
    - Identify general flaws using secure design literature and checklists.
    - Map attack patterns using either the results of abuse case development or a list of historical risks.
    - Identify risks in the architecture based on the use of checklists.
    - Understand and demonstrate the viability of these known attacks.

2- Architectural Risk Analysis (IV)

- **Ambiguity Analysis:**
  - It captures the creative activity required to discover new risks.
  - It requires at least two analysts and experience.
- **Weakness Analysis:**
  - Is aimed at understanding the impact of software dependencies.
  - Software is built on top of complex middleware frameworks.
  - Understand what kind of assumptions we make about outside software.
  - What happens when assumptions fail?

3- Software Penetration Testing

- A method of evaluating software security by simulating an attack in an outside-in manner.
- Testing the system in its final.
- Most commonly applied of all seven touchpoints.
- Use static and dynamic penetration testing tools.
    - Examples: fault injection tools, Cenzic, Holodeck, SPI Dynamics, Immunity CANVAS, and IEInspector HTTP Analyzer.

4- Risk-Based Security Testing (I)

- Testers must follow:
  - Risk-based approach:
    - Grounded in both system's architectural reality and the attacker's mindset.
    - Focused on areas of code where an attack is likely to succeed.
- Different from Penetration Testing:
  - Level of approach: Inside-out approach.
  - Timing of testing: Before the software is complete.
- Should start prior to system integration.
- Risk analysts should identify and rank risks.

4- Risk-Based Security Testing (II)

- Security Testing must involve two approaches:
  - **Functional security testing:**
    - Testing mechanisms that ensure that functionality is well implemented.
  - **Adversarial security testing:**
    - Risk-based security testing motivated by understanding the attacker's approach.

5- Abuse Cases (I)

- Determining what the software *can't* and *won't* do (No positive features).
  - Example: users can't enter more than 50 characters.
- Must anticipate abnormal behaviors.
- Don't accept the idea that says "No one would do these things".
- What happens when functional security mechanisms are compromised?

5- Abuse Cases (II)

- **Generating abuse cases:**
  - **Creating Anti-Requirements:**
    - Provide insights into how attackers will approach system's assumptions.
    - Applied throughout the lifecycle.
    - Generated by security analysts.
  - **Creating an attack model:**
    - Explicit consideration of well-known attacks.
    - Select attack patterns relevant to the system.
    - Build abuse cases around attack patterns.
    - Include anyone who can gain access to the system.

6- Security Requirements

- Cover both functional security and emergent characteristics.
- Satisfy three criteria:
  - **Definition:** Must be explicitly defined what security requirements are.
  - **Assumption:** Must take into account the assumptions that the system will behave as expected.
  - **Satisfaction:** Security requirements must satisfy the security goals, and the system must satisfy the security requirements.

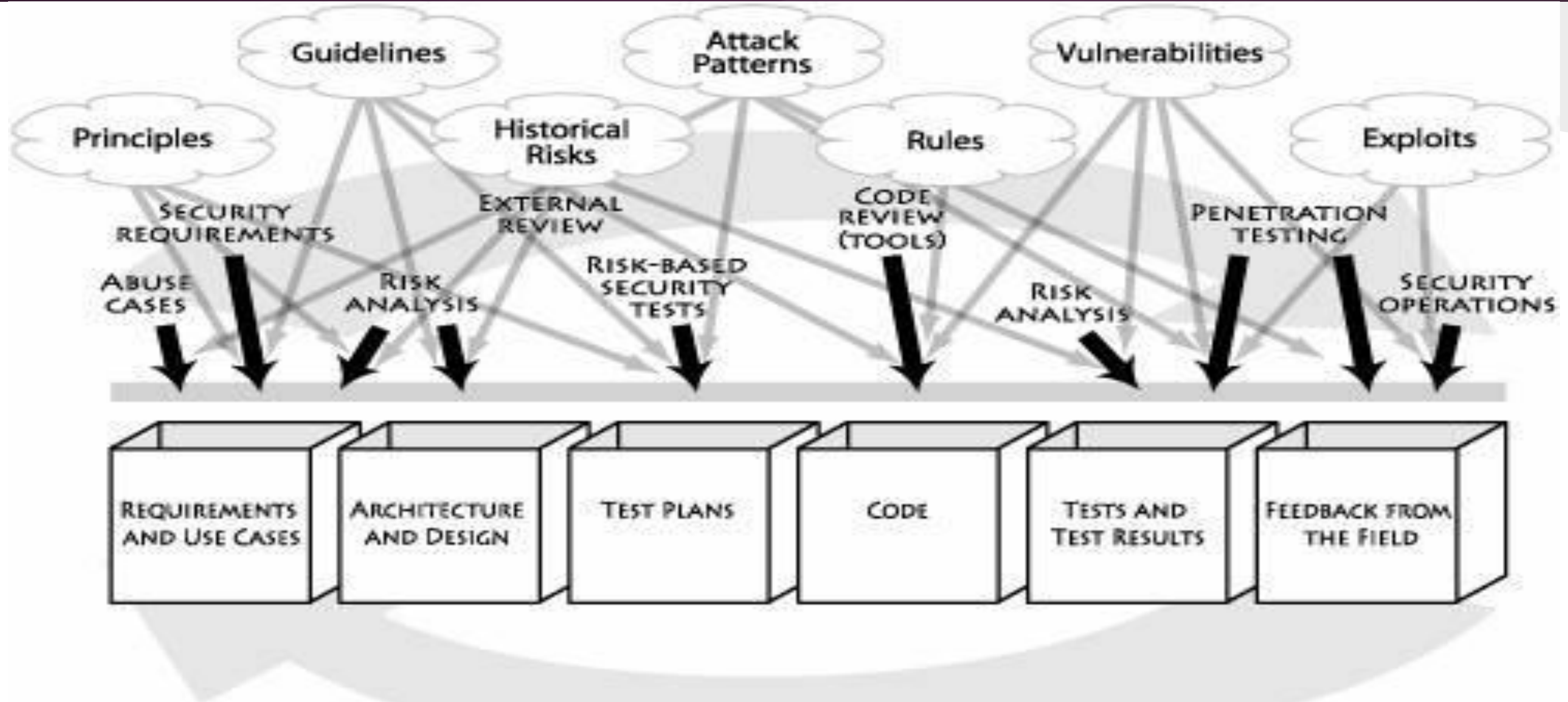
7- Security Operations

- Software security operations can be integrated with network security operations.
- Defensive technique:
  - Understanding software behaviour that leads into successful attack.
- Knowledge gained by understanding attacks should be cycled back into software development.

Pillar III: Knowledge (I)

- Involves gathering, encapsulating, and sharing security knowledge.
- Software Security Knowledge Catalogs:
  - Principles.
  - Guidelines.
  - Rules.
  - Vulnerabilities.
  - Exploits.
  - Attack Patterns.
  - Historical Risks.

Pillar III: Knowledge (II)



Mapping of software security knowledge catalogs to various software artifacts and software security best practices.

Part 2:

19 Deadly Sins (defects)

19 Deadly Sins (defects)

1. Buffer Overruns
2. Format String problems
3. Integer Overflows
4. SQL Injection
5. Command Injection
6. Failing to Handle Errors
7. Cross-Site Scripting
8. Failing to Protect Network Traffic
9. Use of "magic" URLs and Hidden Forms
10. Improper use of SSL and TLS
11. Use of Weak Password-Based Systems
12. Failing to Store and Protect Data Securely
13. Information Leakage
14. Improper File Access
15. Trusting Network Name Resolution
16. Race Conditions
17. Unauthenticated Key Exchange
18. Cryptographically Strong Random Numbers
19. Poor Usability

19 Deadly Sins

Sin 1: Buffer Overruns “smashing the stack”

- The core problem is that user data and program flow control information are intermingled for the sake of performance, and low-level languages allow direct access to application memory.
  - Mostly prevalent in C and C++.
- Occurs when a program allows input to write beyond the end of the allocated buffer.
- The effect of a buffer overrun is anything from a crash to the attacker gaining complete control of the application.

19 Deadly Sins

Sin 1: Buffer Overruns examples

```
void DontDoThis(char *input)
{
        char buf[16];
        strcpy(buf, input);
        printf("%s\n", buf);
}

int main(int argc, char *argv[])
{
        DontDoThis(argv[1]);
        return 0;
}
```

Morris finger worm:

```
    char buf[20];
    gets(buf);
```

```
bool CopyStructs(InputFile *pInFile, unsigned long count)
{
        unsigned long i;
        m_pStructs = new Structs[count];
        for (i=0; i<count; i++)
        {
                if (!ReadFromFile(pInFile, &(m_pStructs[i])))
        break;
        }
}
```

```
#define MAX_BUF 256
void BadCode(char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);

    //of course we can use strcpy safely
    if(len < MAX_BUF)
        strcpy(buf, input);
}
```

If a short is 2 bytes and input > 32767, then len becomes a negative number

If input is not null-terminated/

Slightly better: Use size_t to define size for MAX_BUF and len

19 Deadly Sins

Sin 1: Buffer Overruns: Spotting the sin pattern

Here are the components to look for:

- Input, whether read from the network, a file, or from the command line
- Transfer of data from said input to internal structures
- Use of unsafe string handling calls
- Use of arithmetic to calculate an allocation size or remaining buffer size

Redemption Steps:

- Replace Dangerous String Handling Functions
- Audit Allocations
- Check Loops and Array Accesses
- Replace C String Buffers with C++ Strings
- Replace Static Arrays with STL Containers
- Use Analysis Tools

19 Deadly Sins

Sin 2: Format String Problems

- The root cause of format string bugs is trusting user-supplied input without validation.
  - Mostly prevalent in C and C++.
- Can also occur when the format strings are read from an untrusted location the attacker controls.
- Even if you're not dealing with C/C++
  - Users can be misled
  - An attacker might also launch cross-site scripting or SQL injection attacks

19 Deadly Sins

- Sin 2: Format String Problems examples

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    if(argc > 1)
        printf(argv[1]);

    return 0;
}
```

bug.exe

%x %x → 12ffc0 4011e5

The %x specifier
reads the stack
4 bytes at a time
and outputs them

Leaks important
info to the attacker

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Input is %d characters long.\n", bytes);
```

Usage:
bug.exe "Hello"
Hello
Input is 5 characters long

The %n specifier writes
4 bytes at a time based on the length
of the previous argument
Carefully crafted, allows an attacker
to place own data into the stack

19 Deadly Sins

Sin 2: Format string patterns : Spotting the sin pattern

In C/C++, look for functions from the printf family. Problems to look for are:

- printf(user_input);
- fprintf(STDOUT, user_input);

Redemption Steps:

- The first step is never pass user input directly to a formatting function, and also be sure to do this at every level of handling formatted output
- C/C++ Redemption: printf("%s", user_input);

19 Deadly Sins

Sin 3: Integer Overflows

- The core of the problem is that for nearly every binary format we can choose to represent numbers, there are operations where the result isn't what you'd get with pencil and paper.
- There are exceptions—some languages implement variable-size integer types, but these are not common and do come with some overhead.
- C and C++ have true integer types; and modern incarnations of Visual Basic pack all the numbers into a floating point type known as a "Variant," so you can declare an int, divide 5 by 4, and expect to get 1. Instead, you get 1.25.

19 Deadly Sins

Sin 3: Integer Overflows examples

```
const long MAX_LEN = 0x7FFF;

short len = strlen(input);

if (len < MAX_LEN)
{
    // Do stuff
}
```

If a short is 2 bytes and input > 32767, then len becomes a negative number

Overflow Problem in C#

```
byte a, b;
a = 255;
b = 1;
byte c = (a + b);
ERROR: Cannot implicitly convert type 'int' to 'byte'
```

Problem here to detect whether two unsigned 16-bit numbers would overflow
when added?

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
    if (x + y < x)
        return false;
    return true;
}
```

19 Deadly Sins

Sin 3: Integer Overflows: Spotting the sin pattern

- Anything doing arithmetic
- Especially if input provided by the user
- Focus especially on array index calculations

Redemption Steps:

- Use size_t type in C/C++
- Use unsigned integers if appropriate, easier to verify
- Avoid "clever" code in favor of straightforward code

19 Deadly Sins

Sin 4: SQL Injection

- Perhaps the greatest risk is a SQL injection attack where the attacker gains private PII or sensitive data.

  - Break into server using exploit like buffer overrun
  - Go through open port with sysadmin password
  - Social engineering
  - SQL injection attacks

- The damage is not limited to the data in the database; a SQL injection attack could lead to server, and potentially network, compromise also. For an attacker, a compromised back-end database is simply a stepping stone to bigger and better things.

19 Deadly Sins

Sin 4: SQL Injection examples

```
using System.Data;
using System.Data.SqlClient;
...
string ccnum = "None";
try {
    SqlConnection sql= new SqlConnection(
        @"data source=localhost;" +
        "user id=sa;password=pAs$w0rd;");
    sql.Open();
    string sqlstring="SELECT ccnum" +
        " FROM cust WHERE id=" + Id;
    SqlCommand cmd = new SqlCommand(sqlstring,sql);
    ccnum = (string)cmd.ExecuteScalar();
    } catch (SqlException se) {
        Status = sqlstring + " failed\n\r";
        foreach (SqlError e in se.Errors) {
            Status += e.Message + "\n\r";
    }
} catch (SqlException e) {
    // OOops!
}
```

```
import java.*;
import java.sql.*;
...
public static boolean doQuery(String Id) {
    Connection con = null;
    try
    {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                    "//localhost:1433", "sa", "$3cre+");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(" SELECT ccnum FROM cust WHERE id = " +
Id);
        while (rs.next()) {
            // Party on the query results
        }
        rs.close();
        st.close();
    }
catch (SQLException e)
    {
        // OOPS!
        return false;
    }
    catch (ClassNotFoundException e2)
    {
        // Class not found
        return false;
    }
    finally
    {
        try
        {
            con.close();
        } catch(SQLException e) {}
    }   return true;
}
```

19 Deadly Sins

Sin 4: SQL Injection: Spotting the sin pattern

- Takes user input
- Does not check user input for validity
- Uses user-input data to query a database
- Uses string concatenation or string replacement to build the SQL query or uses the SQL exec command.

Redemption Steps:

- The simplest and safest redemption steps are to never trust input to SQL statements, and to use prepared or parameterized SQL statements, also known as *prepared statements*.
- Filter any unallowable characters like ‘ or “”
- Never use string concatenation to build SQL statements

19 Deadly Sins

Sin 5: Command Injection

- User input that was meant to be data actually can be partially interpreted as a command of some sort.
- Often, that command can give the person with control over the data access to far more access than was ever intended.
- Problems that occur when untrusted data is placed into data that is passed to some sort of compiler or interpreter, where the data might, if it's formatted in a particular way, be treated as something other than data.

19 Deadly Sins

Sin 5: Command Injection examples

```
char buf[1024];
snprintf(buf, "system lpr -P %s", user_input, sizeof(buf)-1);
system(buf);
```

```
def call_func(user_input, system_data):  exec
'special_function_%s("%s")' % (system_data, user_input)
```

```
special_function_sample("fred")
```

```
fred"); print ("foo
```

```
special_function_sample("fred"); print ("foo")
```

19 Deadly Sins

Sin 5: Command Injection: Spotting the sin pattern

- Commands (or control information) and data are placed inline next to each other.
- There is some possibility that the data might get treated as a command, often due to characters with special meanings, such as quotes and semicolons.
- Control over commands would give users more privileges than they already have.

Redemption Steps:

- Check the data to make sure it is okay.
- Take an appropriate action when the data is invalid.

19 Deadly Sins

Sin 6: Failing to Handle Errors

- Sometimes a program can end up in an insecure state, but more often the result is a denial of service issue as the application simply dies.
- The unfortunate reality is that any reliability problem in a program that leads to the program crashing, aborting, or restarting is a denial of service issue, and therefore can be a security problem, especially for server code.
- A common source of errors is sample code that has been copied and pasted. Often sample code leaves out error return checking to make the code more readable.

19 Deadly Sins

Sin 6: Failing to Handle Errors Variants

- Yielding too much information
- Ignoring errors
- Misinterpreting errors
- Using useless error values
- Handling the wrong exceptions
- Handling all exceptions

19 Deadly Sins

Sin 6: Failing to Handle Errors: Spotting the sin pattern

- A code review is by far the most efficient way to spot these.

Redemption Steps:

- Handle the appropriate exceptions in your code.
- Don’t “gobble” exceptions.
- Make sure you check return values when appropriate.

19 Deadly Sins

Sin 7: Cross-Site Scripting

- A form of security defect unique to web-based applications that allow user data tied to the vulnerable web server's domain, usually held in cookies, to be disclosed to a malicious third party.
    - The cookie is transferred from a client computer accessing a valid, but vulnerable, web-server site to a site of the attacker's bidding.
- Malicious user can inject script code that is then executed when another user views that page.

19 Deadly Sins

Sin 7: Cross-Site Scripting working

1. The attacker identifies a web site that has one or more XSS bugs—for example, a web site that echoes the contents of a querystring.
2. The attacker crafts a special URL that includes a malformed and malicious querystring containing HTML and script, such as JavaScript.
3. The attacker finds a victim, and gets the victim to click a link that includes the malformed querystring. This could be simply a link on another web page, or a link in an HTML e-mail.
4. The victim clicks the links and the victim's browser makes a GET request to the vulnerable server, passing the malicious querystring.
5. The vulnerable server echoes the malicious querystring back to the victim's browser, and the browser executes the JavaScript embedded in the response.

19 Deadly Sins

Sin 7: Cross-Site Scripting: Spotting the sin pattern

Any application that has the following pattern is at risk of cross-site scripting:

- The web application takes input from an HTTP entity such as a querystring, header, or form.
- The application does not check the input for validity.
- The application echoes the data back into a browser.

Redemption Steps:

- Restrict the input to valid input only. Most likely you will use regular expressions for this.
- HTML encode the output.

19 Deadly Sins

Sin 8: Failing to Protect Network Traffic

- These kinds of attacks are possible because so many network protocols fail to protect network traffic adequately.

- Many important protocols, such as Simple Mail Transfer Protocol (SMTP) for mail relay, Internet Message Access Protocol (IMAP) and Post Office Protocol (POP) for mail delivery, and HyperText Transfer Protocol (HTTP) for web browsing provide no security at all, or at most, provide basic authentication mechanisms that are easily attacked.

19 Deadly Sins

Sin 8: Failing to Protect Network Traffic forms

Network attacks can take a wide variety of forms:

- **Eavesdropping** The attacker listens in to the conversation and records any valuable information, such as login names and passwords. **Replay** The attacker takes existing data from a data stream and replays it. This can be an entire data stream, or just part of one. For example, one might replay authentication information in order to log in as someone else, and then begin a new conversation.
- **Spoofing** The attacker mimics data as if it came from one of the two parties.
- **Tampering** The attacker modifies data on the wire, perhaps doing something as innocuous as changing a 1 bit to a 0 bit.
- **Hijacking** The attacker waits for an established connection, and then cuts out one of the parties, spoofing the party's data for the rest of the conversation.

19 Deadly Sins

Sin 8: Failing to Protect Network Traffic: Spotting the sin pattern

- An application uses a network
- Designers overlook or underestimate network-level risks

Redemption Steps:

- Use SSL/TLS for any network connections, if at all possible, or else some other abstraction, such as Kerberos.
- Basic security services on the wire:
    - confidentiality, initial authentication, and ongoing authentication.

19 Deadly Sins

Sin 9: Use of Magic URLs and Hidden Form Fields

- There’s nothing stopping a user from looking at the source content, and then sending an “updated” form with a changed data(using Perl, for example) back to the server. Hidden fields are not really hidden at all.

- “Magic URLs”: many web-based applications carry authentication information or other important data in URLs.

    - In some cases, this data should not be made public, because it can be used to hijack or manipulate a session.
    - In other cases, Magic URLs are used as an ad hoc form of access control, as opposed to using credential-based systems.

19 Deadly Sins

- Sin 9: Use of Magic URLs and Hidden Form Fields examples

Hidden Form Fields to pass variables

```
<form action = " /"
    <input type=text name="product">
    <input type=hidden name="price" value="300">
</form>
```

Magic URL

www.xyzzy.com?id=TXkkZWNyZStwQSQkdzByRA==

“base64 decoder yields “My$ecre+pA$$w0rD.”

19 Deadly Sins

Sin 9: Use of Magic URLs and Hidden Form Fields: Spotting the sin pattern

- Sensitive information is read by the web app from a form or URL.
- The data is used to make security, trust, or authorization decisions.
- The data is provided over an insecure or untrusted channel.

Redemption Steps:

Consider the following threats:

  - An attacker views the data
  - An attacker replays the data
  - An attacker predicts the data
  - An attacker changes the data

- Use SSL or store data on server side
- Session variables, encrypted

19 Deadly Sins

Sin 10: Improper Use of SSL and TLS

- Proper server authentication doesn't usually happen automatically. In fact, it often requires writing a lot of code.
- When server authentication isn't done properly, an attacker can eavesdrop, or modify or take over conversations, generally without being detected.

Examples:

- Most mailers lack support for CRLs or OCSP.
- Proper validation when using a web browser requires a user to click on the lock and look at the certificate, to make sure the hostname in the certificate matches the hostname the user intended.
- By default, Stunnel does no validation. If you do request validation, you can either validate optionally, validate against an allow list, or validate the date and chain of trust, without actually checking the proper certificate fields.

19 Deadly Sins

Sin 10: Improper Use of SSL and TLS: Spotting the sin pattern

- SSL or TLS is used, and
- HTTPS is not used, and
- The library or client application code fails to check whether the server certificate is endorsed by a known CA, or
- The library or client application code fails to validate the specific data within the server certificate.

Redemption Steps:

When it's reasonable to use SSL or TLS, do so, making sure that the following is true:

- You're using the most recent version of the protocol (as of this writing, it's TLS 1.1).
- You use a strong cipher suite (particularly not one with RC4 or DES).
- You validate that the certificate is within its validity period.
- You ensure that the certificate is endorsed by a trusted source (root CA), either directly or indirectly.
- You validate that the hostname in the certificate matches the one you expect to see.

Also, you should try to implement one or more revocation mechanisms, particularly basic CRL checking or OCSP.

19 Deadly Sins

Sin 11: Use of Weak Password-Based Systems

- Security experts hate passwords because people will use their kids' names as passwords, or else write them down and stick them under the keyboard if they're forced to use stronger passwords.

- In some respect, any software system using passwords is a security risk.

Guidelines for Password Resets

- Locking users out of accounts for too many bad
- password attempts may result in DoS

19 Deadly Sins

Sin 11: Use of Weak Password-Based Systems examples

TENEX Operating System pseudocode to validate:

```
For i = 0 to len(typed_password)
     if i >= len(actual_password) fail;
     if typed_password[i] != actual_password[i] fail;
     if i < len(actual_password) fail;
Success;
```

Flaw: Attacker could put candidate password in memory overlapping page boundaries. First letter on one page, second letter on the next, if the first letter was correct there was a pause while the page for the second letter loaded

19 Deadly Sins

Sin 11: Use of Weak Password-Based Systems: Spotting the sin pattern

- Is a program using traditional or handmade password systems without using some other authentication technique to provide defense in depth?
- Even if there is multifactor authentication, there can still be some risks anytime you're using a password system, such as account lock-out due to failed login attempts.

Redemption Steps:

- Multifactor Authentication
- Storing and Checking Passwords
- Guidelines for Choosing Protocols
- Guidelines for Password Resets
- Guidelines for Password Choice
- Other Guidelines

19 Deadly Sins

Sin 12: Failing to Store and Protect Data Securely

- There are a number of aspects you need to consider when storing data securely:
    - permissions required to access the data, data encryption issues, and threats to stored secrets.
- Many developers hardcode secret data into software, such as cryptographic keys and passwords, that they do not expect users to recover, believing that reverse engineering is too difficult to do.

19 Deadly Sins

Sin 12: Failing to Store and Protect Data Securely examples

CVE-2000-0100

- The SMS Remote Control program is installed with insecure permissions, which allows local users to gain privileges by modifying or replacing the program.

CAN-2004-0391

- A default username/password pair is present in all releases of the Cisco Wireless Lan Solution Engine. A user that logs in using this username has complete control of the device. This username cannot be disabled. There is no workaround.

19 Deadly Sins

Sin 12: Failing to Store and Protect Data Securely: Spotting the sin pattern

Look for code that:

- Sets access controls
- AND grants write access to low-privileged users
- Creates an object without setting access controls
- AND creates the object in a place writable by low-privileged users
- Writes configuration information into a shared area
- Writes sensitive information into an area readable by low-privileged users

Redemption Steps:

- The best approach to solving design-level problems is to use threat modeling.
- Educate yourself about the platform(s) that you write code for, and understand how the underlying security subsystems really work.
- Make a distinction between system-wide information and user-level information.
- Remediation of embedded secrets:
  - Use the operating system’s security technologies
  - Move the secret data out of harm’s way

19 Deadly Sins

Sin 13: Information Leakage

At a high level, there are two primary ways in which information gets leaked:

- **By accident** The data is considered valuable, but it got out anyway, perhaps due to a logic problem in the code, or perhaps through a non-obvious channel. Or the data would be considered valuable if the designers were to recognize the security implications.

- **By intention** Usually the design team has a mismatch with the end user as to whether data should be protected. These are usually privacy issues.

19 Deadly Sins

Sin 13: Information Leakage examples

Dan Bernstein's AES Timing Attack

- Dan Bernstein was able to perform a remote timing attack against the OpenSSL 0.9.7 implementation of AES.

CAN-2005-1411

- ICUII is a tool for performing live video chat. Version 7.0.0 has a bug that allows an untrusted user to view passwords due to a weak access control list (ACL) on the file that allows everyone to read the file.

CAN-2005-1133

- This defect in IBM's AS/400 is a classic leakage; the problem is that different error codes are returned depending on whether an unsuccessful login attempt to the AS/400 POP3 server is performed with a valid or invalid username.

19 Deadly Sins

Sin 13: Information Leakage: Spotting the sin pattern

- A process sending output to users that comes from the OS or the run-time environment
- Operations on secret data that don’t complete in a fixed amount of time, where the time is dependent on the makeup of the secret data
- Accidental use of sensitive information
- Unprotected or weakly protected sensitive or privileged data
- Sensitive data sent from a process to potentially low-privileged users
- Unprotected and sensitive data sent over insecure channels

Redemption Steps:

- For straightforward information leakage, the best starting remedy is to determine who should have access to what, and to write it down as a policy your application designers and developers must follow.
- Other defensive techniques are encryption (with appropriate key management, of course) and digital rights management (DRM).

19 Deadly Sins

Sin 14: Improper File Access

There are three common security issues.

- The first is a race condition: after making security checks on a file, there is often a window of vulnerability between the time of check and the time of use (TOCTOU).
- The second common security issue is the classic “it isn’t really a file” problem, where your code opens a file thinking the code is opening a simple file on the disk, but it is, in fact, a link to another file or a device name or a pipe.
- The third common security issue is giving attackers some control over the filename that they shouldn’t have, allowing them to read and potentially write sensitive information.

19 Deadly Sins

Sin 14: Improper File Access examples

```
void AccessFile(char *szFileNameFromUser) {
    HANDLE hFile =
        CreateFile(szFileNameFromUser,
            0,0,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);
    ...
```

```
import os
def safe_open_file(fname, base="/var/myapp"):
    # Remove '..' and '.'
    fname = fname.replace('../', '')
    fname = fname.replace('./', '')
    return open(os.path.join(base, fname))
```

19 Deadly Sins

Sin 14: Improper File Access: Spotting the sin pattern

- It accesses files based on filenames beyond your control.
- You access files using solely filenames and not file handles or file descriptors.
- You open temporary files in public directories, where the temporary filename is guessable.

Redemption Steps:

- If you can keep all files for your application in a place that attackers cannot control under any circumstances.
- Never use a filename for more than one file operation; pass a handle or file descriptor from the first operation to successive function calls.
- Resolve the path to the file you're going to access, following symbolic links and any backwards traversals before performing validation on the path.
- If you insist on opening a temporary file in a public directory, the most reliable way is to take eight bytes from the system cryptographic random number generator, base64 encode it, replace the "/" character that the encoding might output with a "," or some other benign character, and then use the result in your filename.
- Where appropriate (read: if in doubt), lock the file when it is first accessed or created by your code.
- If you know the file will be new and zero bytes in size, then truncate it. This way an attacker cannot give you a prepopulated rogue temporary file.
- Never trust a filename not under your direct control.
- Check whether the file is a real file, not a pipe, a device, or a symlink.

19 Deadly Sins

Sin 15: Trusting Network Name Resolution

- The real problem here is that most developers don't realize how fragile name resolution is, and how easily it is attacked.

- Although the primary name resolution service is DNS for most applications, it is common to find Windows Internet Name Service (WINS) used for name resolution on large Windows networks.

- Although the specifics of the problem vary depending on what type of name resolution service is being used, virtually all of them suffer from the basic problem of not being trustworthy.

19 Deadly Sins

Sin 15: Trusting Network Name Resolution examples

CVE-2002-0676

- SoftwareUpdate for MacOS 10.1.x does not use authentication when downloading a software update, which could allow remote attackers to execute arbitrary code by posing as the Apple update server via techniques such as DNS spoofing or cache poisoning and supplying Trojan Horse updates.

19 Deadly Sins

Sin 15: Trusting Network Name Resolution: Spotting the sin pattern

- This sin applies to any application that behaves as a client or server on the network where the connections are authenticated, or when there is any reason to need to know with certainty what system is on the other end of the connection.

- Using SSL (or to be precise, SSL/TLS) is a good way to authenticate servers, and if your client is a standard browser, the supplier of the browser has done most of the work for you. If your client isn't a standard browser, you must check for two things: whether the server name matches the certificate name, and whether the certificate has been revoked. One little-known feature of SSL is that it can also be used to authenticate the client to the server.

Redemption Steps:

- Ensure that connections are running over SSL.

- Use IPSec.

- If the application is critical, then the most secure way to approach the problem is to use public key cryptography, and to sign the data in both directions.

19 Deadly Sins

Sin 16: Race Conditions

- When two different execution contexts, whether they are threads or processes, are able to change a resource and interfere with one another.
- The typical flaw is to think that a short sequence of instructions or system calls will execute atomically, and that there’s no way another thread or process can interfere.
- Most system calls end up executing many thousands (sometimes millions) of instructions, and often they won’t complete before another process or thread gets a time slice.

19 Deadly Sins

Sin 16: Race Conditions examples

```
list<unsigned long> g_TheList;

unsigned long GetNextFromList()
{
    unsigned long ret = 0;
    if(!g_TheList.empty())
    {
        ret = g_TheList.front();
        g_TheList.pop_front();
    }
    return ret;
}
```

```
char* tmp;
FILE* pTempFile;

tmp = _tempnam("/tmp", "MyApp");
pTempFile = fopen(tmp, "w+");
```

19 Deadly Sins

Sin 16: Race Conditions: Spotting the sin pattern

- More than one thread or process must write to the same resource. The resource could be shared memory, the file system (for example, by multiple web applications that manipulate data in a shared directory), other data stores like the Windows registry, or even a database. It could even be a shared variable!
- Creating files or directories in common areas, such as directories for temporary files (like /tmp and /usr/tmp in UNIX-like systems).
- Signal handlers.
- Non-reentrant functions in a multithreaded application or a signal handler. Note that signals are close to useless on Windows systems and aren't susceptible to this problem.

Redemption Steps:

- Understand how to correctly write reentrant code.
- Whether through forked processes or threads, you need to carefully guard against both the lack of locking shared resources, and incorrectly locking resources.
- If you're executing a signal handler or exception handler, the only really safe thing to do may be to call exit().

19 Deadly Sins

Sin 17: Unauthenticated Key Exchange

- The problem is that key exchange also has security requirements: the exchanged key needs to be secret, and, more importantly, the messages in the protocol need to be properly authenticated.

- The fundamental problem is failing to realize that the connection is insufficiently authenticated (and sometimes not authenticated at all).

- In almost every circumstance, it doesn't make sense to do a key exchange without authentication.

  - All modern authentication protocols intended to be used over a network are also key exchange protocols.

  - Nobody builds key exchange protocols to stand on their own anymore since authentication is a core requirement.

19 Deadly Sins

Sin 17: Unauthenticated Key Exchange examples

- Man-in-the-middle attacks are pretty well known, and we've seen this problem repeatedly in "real-world" systems that were built by starting with books and then trying to build a cryptosystem from that.

- In February of 2001, BindView discovered a man-in-the-middle attack on Novell's Netware where they were improperly authenticating a home-made key exchange/authentication protocol. Their home-made protocol used an RSA-based scheme for key exchange instead of Diffie-Hellman. They attempted to authenticate by using a password-based protocol, but did not properly authenticate the key exchange messages themselves. The password protocol was encrypted with the RSA keys, but the password wasn't used to validate that the keys were owned by the right parties. An attacker could spoof the server, in which case the client would public-key encrypt a password validator to the attacker. Then, the attacker could replay that validator to the server, which would succeed, allowing the attacker to be a man in the middle.

19 Deadly Sins

Sin 17: Unauthenticated Key Exchange: Spotting the sin pattern

- This can occur anytime an application performs authentication over the network where the connection establishment requires some sort of cryptography to provide authentication.

Redemption Steps:

- Protocols such as SSL/TLS or Kerberos.
- Make sure that the resulting key exchange is used to provide ongoing authentication services.
- Don’t design your own protocol.
- If you have a preexisting protocol that is custom-built, consider migrating to an off-the-shelf solution, where the set of things that could go wrong are small and well understood, such as SSL/TLS.

19 Deadly Sins

Sin 18: Cryptographically Strong Random Numbers

- The biggest sin you can commit with random numbers is not using them when they should be used.

- Three different kinds of random numbers:

  - Non-cryptographic pseudo-random number generators (noncrytographic PRNG)
  - Cryptographic pseudo-random number generators (CRNGs)
  - “True” random number generators (TRNGs), which are also known as *entropy generators*

19 Deadly Sins

Sin 18: Cryptographically Strong Random Numbers examples

- In 1996, grad students Ian Goldberg and David Wagner determined that Netscape's SSL implementation was creating "random" session keys by applying Message Digest 5 (MD5) to some not-very-random data, including the system time and the process ID. As a result, they could crack real sessions in less than 25 seconds on 1996 hardware. This takes less than a fraction of a second today.

- Really old versions of OpenSSL relied on the user to seed the PRNG, and would give only this warning: "Random number generator not seeded!!!" Some people just ignored it, and the program would go on its merry way. Other people would seed with a constant string, and the program would go on its merry way.

19 Deadly Sins

Sin 18: Cryptographically Strong Random Numbers: Spotting the sin pattern

- The sin can manifest anytime you have the need to keep data secret, even from someone who guesses. Whether you're encrypting or not, having good random numbers is a core requirement for a secure system.

Redemption Steps:

- For the most part, you should use the system CRNG. The only exceptions are when you're coding for a system that doesn't have one, when you have a legitimate need to be able to replay number streams, or when you need more security than the system can produce (particularly if you're generating 192-bit or 256-bit keys on Windows using the default cryptographic provider).

19 Deadly Sins

Sin 19: Poor Usability

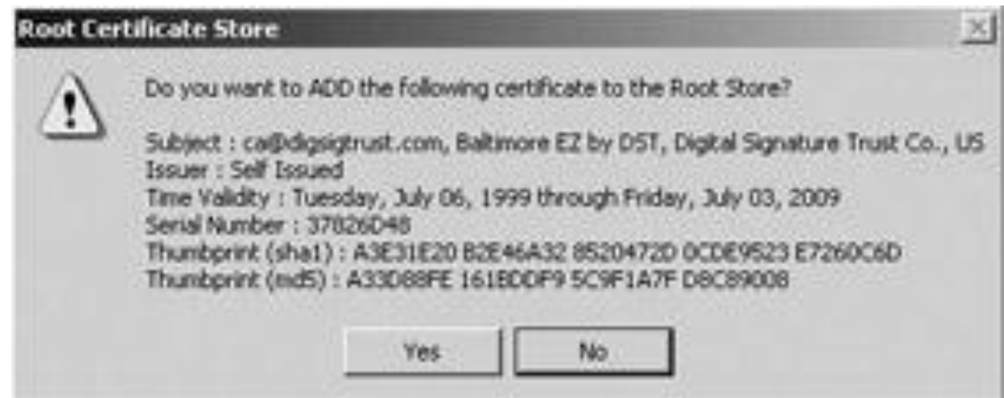
- The second law from the 10 Immutable Laws of Security Administration states that "Security only works if the secure way also happens to be the easy way."

It is common to see security-related text and messages exhibiting one or more of the following properties:

- Too little appropriate information: This is the bane of the administrator: not enough information to make a good security decision.
- Too much information: This is the bane of the normal user: too much security information that is simply confusing.
- Too many messages: Eventually both admins and users will simply click the "OK" or "Yes" buttons when faced with too many messages. And that last acknowledgment may be the wrong thing to do.
- Inaccurate or generic information: There is nothing worse than this because it doesn't tell the user anything. Of course, you don't want to tell an attacker too much either; it's a fine balance.
- Errors with only error codes: Error codes are fine, so long as they are for the admins' benefit, and they include text to help the user.

19 Deadly Sins

Sin 19: Poor Usability examples



19 Deadly Sins

Sin 19: Poor Usability: Spotting the sin pattern

- The pattern here is a failure to explore the way the typical user is going to interact with your security features.
- Poor usability can also mean poor security
  - Always clicking "OK" when given lots of dialogs
  - Cryptic error or status messages

Redemption Steps:

- When users are involved, make the UI simple and clear
- Make security decisions for users
- Make selective relaxation of security policy easy
- Clearly indicate consequences
- Make it actionable
- Provide central management

19 Deadly Sins

Sin 1: Buffer Overruns Summary

- Do carefully check your buffer accesses by using safe string and buffer handling functions.
- Do use compiler-based defenses such as /GS and ProPolice.
- Do use operating-system-level buffer overrun defenses such as DEP and PaX.
- Do understand what data the attacker controls, and manage that data safely in your code.
- Do not think that compiler and OS defenses are sufficient—they are not; 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.

19 Deadly Sins

Sin 2: Format String Problems Summary

- Do use fixed format strings, or format strings from a trusted source.
- Do check and limit locale requests to valid values.
- 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.

19 Deadly Sins

Sin 3: Integer Overflows Summary

- Do check all calculations used to determine memory allocations to check thatthe arithmetic cannot overflow.
- Do check all calculations used to determine array indexes to check that the arithmetic cannot overflow.
- Do use unsigned integers for array offsets and memory allocation sizes.
- Do not think languages other than C/C++ are immune to integer overflows.

19 Deadly Sins

Sin 4: SQL Injection Summary

- Do understand the database you use. Does it support stored procedures? What is the comment operator? Does it allow the attacker to call extended functionality?
- Do check the input for validity and trustworthiness.
- Do use parameterized queries, also known as prepared statements, placeholders, or parameter binding to build SQL statements.
- Do store the database connection information in a location outside of the application, such as an appropriately protected configuration file or the Windows registry.
- Consider removing access to all user-defined tables in the database, and granting access only through stored procedures. Then build the query using stored procedure and parameterized queries.

19 Deadly Sins

Sin 5: Command Injection Summary

- 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 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 input validation; instead, write simple and clear validators by hand.

19 Deadly Sins

Sin 6: Failing to Handle Errors Summary

- Do check the return value of every security-related function.
- Do check the return value of every function that changes a user setting or a machine-wide setting.
- Do make every attempt to recover from error conditions gracefully, to help avoid denial of service problems.
- Do not catch all exceptions without a very good reason, as you may be masking errors in the code.
- Do not leak error information to untrusted users.

19 Deadly Sins

Sin 7: Cross-Site Scripting Summary

- Do check all web-based input for validity and trustworthiness.
- Do HTML encode all output originating from user input.
- Do not echo web-based input without checking for validity first.
- Do not store sensitive data in cookies.
- Consider using as many extra defenses as possible.

19 Deadly Sins

Sin 8: Failing to Protect Network Traffic Summary

- Do perform ongoing message authentication for all network traffic your application produces.
- Do use a strong initial authentication mechanism.
- Do encrypt all data for which privacy is a concern. Err on the side of privacy.
- Do use SSL/TLS for all your on-the-wire crypto needs, if at all possible. It works!
- Do not ignore the security of your data on the wire.
- Do not hardcode keys, and don't think that XORing with a fixed string is an encryption mechanism.
- Do not hesitate to encrypt data for efficiency reasons. Ongoing encryption is cheap.
- Consider using network-level technologies to further reduce exposure whenever it makes sense, such as firewalls, VPNs, and load balancers.

19 Deadly Sins

Sin 9: Use of Magic URLs and Hidden Form Fields Summary

- Do test all web input, including forms, with malicious input.
- Do understand the strengths and weaknesses of your approach if you're not using cryptographic primitives to solve some of these issues.
- Do not embed confidential data in any HTTP or HTML construct, such as the URL, cookie, or form, if the channel is not secured using an encryption technology such as SSL, TLS, or IPSec, or it uses application-level cryptographic defenses.
- Do not trust any data, confidential or not, in a web form, because malicious users can easily change the data to any value they like, regardless of SSL use or not.
- Do not think the application is safe just because you plan to use cryptography; attackers will attack the system in other ways. For example, attackers won't attempt to guess cryptographically random numbers; they'll try to view it.

19 Deadly Sins

Sin 10: Improper Use of SSL and TLS Summary

- Do use the latest version of SSL/TLS available, in order of preference: TLS 1.1, TLS 1.0, and SSL3.
- Do use a certificate allow list, if appropriate.
- Do ensure that, before you send data, the peer certificate is traced back to a trusted CA and is within its validity period.
- Do check that the expected hostname appears in a proper field of the peer certificate.
- Consider using an OCSP responder when validating certificates in a trust chain to ensure that the certificate hasn't been revoked.
- Consider downloading CRLs once the present CRLs expire and using them to further validate certificates in a trust chain.

19 Deadly Sins

Sin 11: Use of Weak Password-Based Systems Summary

- Do ensure that passwords are not unnecessarily snoopable over the wire when authenticating (for instance, do this by tunneling the protocol over SSL/TLS).
- Do give only a single message for failed login attempts, even when there are different reasons for failure.
- Do log failed password attempts.
- Do use a strong, salted cryptographic one-way function based on a hash for password storage.
- Do provide a secure mechanism for people who know their passwords to change them.

19 Deadly Sins

Sin 12: Failing to Store and Protect Data Securely Summary

- Do think about the access controls your application explicitly places on objects, and the access controls objects inherit by default.
- Do realize that some data is so sensitive it should never be stored on a general purpose, production server—for example, long-lived X.509 private keys, which should be locked away in specific hardware designed to perform only signing.
- Do leverage the operating system capabilities to secure secret and sensitive data.
- Do use appropriate permissions, such as access control lists (ACLs) or Permissions if you must store sensitive data.
- Do remove the secret from memory once you have used it.
- Do scrub the memory before you free it.

19 Deadly Sins

Sin 13: Information Leakage Summary

- Do define who should have access to what error and status information data.
- Do use operating system defenses such as ACLs and permissions.
- Do use cryptographic means to protect sensitive data.
- Do not disclose system status information to untrusted users.
- Do not provide high-precision time stamps alongside encrypted data. If you need to provide them, remove precision, and/or stick it in the encrypted payload (if possible).
- Consider using other less commonly used operating system defenses such as file-based encryption.
- Consider using cryptography implementations explicitly hardened against timing attacks.
- Consider using the Bell-LaPadula model, preferably through a preexisting mechanism.

19 Deadly Sins

Sin 14: Improper File Access Summary

- Do be strict about what you will accept as a valid filename.
- Do not blindly accept a filename thinking it represents a valid file—especially on server platforms.
- Consider storing temporary files in the user's temporary directory, not in a shared location. This has an added benefit of making it easier to run your application in least privilege, because the user has full access to their private directory. However, in many cases, only elevated accounts such as administrator and root can access system temporary directories.

19 Deadly Sins

Sin 15: Trusting Network Name Resolution Summary

- Do use cryptography to establish the identity of your clients and servers. A cheap way to do this is through SSL.
- Do not trust DNS information—it isn't reliable!
- Consider specifying IPSec for the systems your application will run on.

19 Deadly Sins

Sin 16: Race Conditions Summary

- Do write code that doesn’t depend on side effects.
- Do be very careful when writing signal handlers.
- Do not modify global resources without locking.
- Consider writing temporary files into a per-user store instead of a world-writable space.

19 Deadly Sins

Sin 17: Unauthenticated Key Exchange Summary

- Do realize that key exchange alone is often not secure. You must authenticate the other party or parties also.
- Do use off-the-shelf solutions for session establishment, such as SSL/TLS.
- Do ensure that you read all the fine print to make sure you have strongly authenticated every party.
- Consider calling in a cryptographer if you insist on using custom solutions.

19 Deadly Sins

Sin 18: Cryptographically Strong Random Numbers Summary

- Do use the system cryptographic pseudo-random number generator (CRNGs) when at all possible.
- Do make sure that any other cryptographic generators are seeded with at least 64 bits of entropy, preferably 128 bits.
- Do not use a noncryptographic pseudo-random number generator (noncrytographic PRNG).
- Consider using hardware random number generators (RNGs) in high-assurance situations.

19 Deadly Sins

Sin 19: Poor Usability Summary

- Do understand your users' security needs, and provide the appropriate information to help them get their jobs done.
- Do default to a secure configuration whenever possible.
- Do provide a simple and easy to understand message, and allow for progressive disclosure if needed by more sophisticated users or admins.
- Do make security prompts actionable.
- Do not dump geek-speak in a big-honking dialog box. No user will read it.
- Do not make it easy for users to shoot themselves in the foot—hide options that can be dangerous!
- Consider providing ways to relax security policy selectively, but be explicit and clear about what the user is choosing to allow.

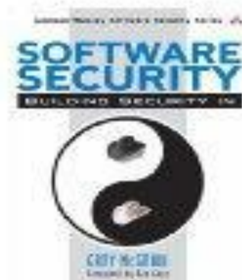
Summary

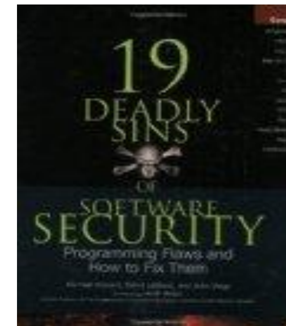
- Fundamentals of software security.
  - Three pillars of software security:
    - Risk Management
    - Touchpoints
    - Knowledge
- Software security design principles.
  1. Code Review
  2. Risk Analysis
  3. Penetration Testing
  4. Risk-Based Security Tests
  5. Abuse Cases
  6. Security Requirements
  7. Security Operations
- Building secure software systems.
  - Learned how to apply best software security practices to various software artifacts throughout the software life cycle in an iterative approach.
- 19 Deadly sins.
  - Learned extensively about the common software vulnerabilities.

Questions?

References

1. Gary McGraw, Software Security: Building Security In. Addison-Wesley Professional, 2006.
2. McGraw, G.; , "Software security," Security & Privacy, IEEE , vol.2, no.2, pp. 80- 83, Mar-Apr 2004doi: 10.1109/MSECP.2004.1281254 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1281254&isnumber=28622
3. Haley, C.B.; Laney, R.; Moffett, J.D.; Nuseibeh, B.; , "Security Requirements Engineering: A Framework for Representation and Analysis," Software Engineering, IEEE Transactions on , vol.34, no.1, pp.133-153, Jan.-Feb. 2008.
4. LeBlanc, John Viega, 19 deadly sins of software security. McGraw-Hill, 2005.





Thank you!