HOW TO CONDUCT A CODE REVIEW

Effective Techniques for Uncovering Vulnerabilities in your code.

Table of Contents

Objectives

By performing the steps in this guide, you will be able to:

- Identify the type of bugs that are important for your code.
- Generate a list of bugs found in the code that should be prioritized for eradication.

Overview

A properly conducted code review can do more for the security of your application than nearly any other step. A large numbers of bugs can be found and fixed before the code makes it into an official build or into the hands of the test team. Additionally the code review process lends itself very well to sharing security best practices amongst a development team and it produces 'lessons' that can be learned from in order to prevent future bugs. Code review is an ongoing process that, ideally, should occur with every code check-in. The cost of high security is eternal vigilance!

The code review approach presented here focuses first on identifying the types of issues that you should look for in the code being reviewed, and then on finding these bugs as quickly and effectively as possible. You will use threat models, architecture diagrams and other inputs in order to guide your review and then can use the list of discovered vulnerabilities to guide future reviews as well as for developer training.

How to Use This Guide

Use this guide in order to conduct an effective code review for security. When using this guide keep the following in mind:

- **Set time limits on your reviews.** While code reviewing it is easy to get lost in the details and lose track of the higher-level security bugs you are looking for. Set a reasonable time limit on your review and then use this to keep yourself from getting stuck. If you find yourself spending too much time in any one place, especially if it is not a high-priority area of code, flag it for later review and move on.
- **Review small chunks of code at a time.** Limit your reviews to small, manageable chunks of code. This will allow you to finish quickly, stay focused, and find a larger number of bugs in the code you are looking at.
- **Review iteratively.** A continuous, iterative approach to code reviewing will allow you to keep your reviews manageable in time and scope. Rather than waiting till the end of a project and reviewing everything at once, review at each check-in. This allows you to focus on what's changed rather than trying to find all the bugs at once.
- **Set clear objectives for your review.** A focused review is an effective review. Spend time at the beginning of your review to understand the bugs that are possible in the code you are

reviewing. Understand the patterns of bad code you want to eradicate and then review with a clear idea of what you are looking for.

- **Understand inputs and outputs for the code you are reviewing.** Dataflow analysis is a powerful mechanism for finding security bugs. Understand every source of data in the code you are reviewing as well as where the data will end up. How much trust you are willing to give the source as well as the ultimate destination of the data both have a major impact on the level of data validation the code should have.
- **Review only for security; look for other problems during another review.** The more you are looking for during a review, the less likely you are to find any of them. Stay focused on the discrete set of security objectives you have for the code review.
- **Review with a partner.** A second set of eyes and the resulting dialog will increase the odds of finding bugs during the review and will keep you fresh longer, extending your effectiveness.

Input

The following input is useful for code review:

- **Code** – You need this to do the review.
- **Architecture diagram** – Use this to understand the high-level workings of the application and where the component you are reviewing fits into the bigger picture.
- **Usage scenarios –** Understand the scenarios that impact the code you will be reviewing.
- **Inputs and outputs –** In order to perform dataflow analysis it is necessary to know each type of input and output the code has. For instance is there a UI, a public API, a socket interface, an XML interface, ODBC connections, files, registry entries, etc.
- **Data flow –** Many times product design will include dataflow diagrams at the component level, however for the purpose of a code review the more granular the better. Ideally you will want to know the path every input takes to every output; you may have to create this as you conduct the review.
- **Use of native code –** It's valuable to know each point at which your managed code will call into native code – interop layers can be a rich source of bugs as the assumptions made while writing managed code will not apply when using native code.
- **Scope –** What is the scope of the review? This can take the form of a list of code that the reviewed code interacts with but is outside the scope of the current review. List assumptions as you review to the edges of these dependencies (these can be checked later) – e.g. We assume that libraryFoo will release the block of memory we pass to it as it promises, we assume the crypto library will wipe the clear text buffer we sent it just after it encrypts it, etc.
- **Reference material –** This can include data schemas if there is interaction with a data store (DB or XML), SDK documentation, comments in the code, and more. Gather anything that can be used to better understand the code as it is being reviewed.
- **Code expert –** Someone you can ask questions of when looking at business logic or code blocks you don't understand.
- **Threat model –** Understand the identified threats and the mitigations that have been implemented for the code you are looking at.

Output

The goal of the code review is to generate a list of vulnerabilities that can be fixed in order to improve security of the code. This list is usually organized by component and will often contain the following information per vulnerability:

- Lines of code that are in error.
- Conditions which will result in an error. For instance is there a particular code path that leads to this vulnerability, or a particular input that must be passed to the function in order for the bug to reveal itself? While it's best if you can prove that the bug can be realized it's not always necessary. Even bugs that are not provably reproducible in the running system are worth fixing. A future code change may alter the path of logic and expose the bug with disastrous consequences.
- Description of what it will take to implement a fix. Is it a single line of code that needs to be altered? Is it a larger architectural problem?

Steps Summary

1. Identify code review objectives
    a. Input
        i. Architecture diagram
        ii. Threat model
        iii. Scope
    b. Output
        i. Code review objectives
2. Review pass #1 - Pick off low hanging fruit with a static analysis scan
    a. Input
        i. Code
        ii. Code review objectives
        iii. Code expert
        iv. Reference material
    b. Output
        i. Vulnerability list (false positives filtered out)
        ii. List of hotspots
3. Review pass #2 – Pick off common security bugs
    a. Input
        i. Code
        ii. Code review objectives

iii. List of hotspots
iv. Code expert
v. Usage scenarios
vi. Input and outputs
vii. Dataflow
viii. Areas that use native code
ix. Reference material

b. Output
i. Vulnerability list

4. Review pass #3 - Look for problems unique to the application architecture
   a. Input
      i. Code
      ii. Code review objectives
      iii. Code expert
      iv. Reference material
   b. Output
      i. Vulnerability list

Code reviewing is not a one-time exercise. Any new code, especially in security sensitive areas, should be code reviewed to discover security vulnerabilities. All vulnerabilities found should not only be placed in a bug database for prioritization and eradication but should be used as input in future code reviews. Over time you can add significantly to the list of bugs you are looking for.

Steps

STEP 1: IDENTIFY CODE REVIEW OBJECTIVES

While it is possible to conduct a code review without knowing what you are looking for, reviews are much more effective with a concrete set of objectives. Code review objectives are a set of bug types that you will be looking for in your application based upon its architecture and the identified threats. For instance it is not important to look for SQL injection bugs if your application has no interactions with a database.

To determine the objectives of your review, consider the following questions:

- **Which of the threats identified in your threat model apply to the code you are reviewing?** After determining which threats apply you can separate the threats into two categories: those that have been mitigated, and those that haven't. Make a list of the mitigated threats, the code written to implement this mitigation is a prime target for security code review.

- **Which common coding errors apply to the code you are reviewing?** Create a list of the technologies used in your application – pay special attention to the architecture to see what other components your application interacts with. Is there a database? Does your component present data on a web page? Does your component interact with native code? Do users supply input to your component, either directly or through an intermediary? This list can then be used to prune the set of bugs you are interested in looking for.
- **What is the scope of your review?** It's important to know what code you *will* be reviewing and what you *won't* be reviewing.

The following are examples of code review objectives:

- Ensure that all un-trusted input to the component is passed to a validation routine before being used.
- Check error handling to make sure that exceptions are caught consistently and close to their source.
- Check calculations whose results are then used for memory allocation or buffer access for numeric overflow or underflow.
- Check cryptographic routines to ensure secrets are cleared quickly.

It's better to conduct multiple short reviews on small chunks of code (e.g. at the time of check-in), however, you may find yourself faced with a large backlog of code to review. In this case it's a good idea to set a time limit on the review. Code reviewing is detailed, tiring work and it is easy to start making mistakes after many hours of review. Also, without a set time limit it is easy to rat-hole and get too deeply into the details of a particular implementation. With a set limit you can force yourself to move on in order to find high-value bugs elsewhere. Another useful trick is to code review in pairs, the resulting dialog and extra set of eyes can keep you fresh much longer than you would manage on your own.

Focused code reviews are effective code reviews. You should look at the code with specific goals, time limits, and knowledge of what bugs you want to uncover. Not only will this substantially raise your chance for success it will also reduce the amount of time you spend reviewing.

STEP 2: CODE REVIEW PASS #1 - PICK OFF LOW HANGING FRUIT WITH A STATIC ANALYSIS SCAN

In this step you use a static analysis tool to find a first set of bugs and improve your understanding of where bugs are likely to be discovered manually.

Not everyone has access to a static analysis tool to aid in their source code review. While this step is not required, we feel it is a valuable step worth documenting. Anything a static analysis tool finds can theoretically be found by manual review as well, however, static analysis tools are unique because they test the code without knowing or requiring any external states to be set. Since the Static Analyzer does not know what the application or function is intended to do it will not make assumptions that a developer or code reviewer might.

Due to its programmatically rigorous nature a static analysis scan may find problems that a manual review will miss, however, the bane of these analysis tools are false positives. While these can be frustrating, the act of reviewing the results of an automated scan can help you gain a better understanding of the code you are reviewing. It forces you to understand *why* a false positive is false which draws you into a deeper understanding of the code including control and dataflow. An additional benefit is that reviewing the results may make bug 'hotspots' more evident. Hotspots are areas of the code that need exceptionally close review due to sloppy code, sensitivity, or both. Static analyzers tend to be good at finding sloppy code such as: Missing error handlers, empty catch blocks, integer overflows and scoping problems. Bugs tend to cluster. If the tool finds a large number of bugs in a particular component or function that area is worth additional manual scrutiny to discover bugs that the scanner may have missed.

Static Analysis Techniques

There are a number of different techniques used by static analysis tools:

Semantic checking

Semantic Analysis allows the analyzer to discover the basic structure and relation of each function within the application. This helps the static analysis tool to better understand how the application will run after build time and to find bugs deeper in the code base. In this part of the analysis an abstract syntax tree can be built to run simulations of each of the functions to calculate how the application will execute.

Strong type checking

This helps ensure the programmer does not make any dangerous type casting assumptions such as accidentally attempting to cast a float or decimal value to an integer type at runtime. This helps ensure round off errors and type conversion errors do not happen at runtime. If the programmer knows that the type conversion will never happen or has taken other cautions to prevent the error then the warning can be safely disregarded.

The static analyzer checks for un-initialized or possibly un-initialized variables. By following the code path from the first declaration of the variable to see if the variable is used before it is assigned to, or if the assigning function may return an invalid type for the variable, the tool can catch possibly un-initialized variables.

Memory allocation checking

Some static analysis tools can check to see how memory is being allocated to find cases where memory is being improperly used. If memory is being allocated but no deallocation can be found this might indicate a memory leak. Deallocating memory that is not allocated within the same function may cause the application to crash if earlier allocation attempts fail. Many static analysis tools also can catch bugs due to mismatched memory sizes. This type of checking is useful for native code only, not managed code.

Logical statement checking

This feature allows the Static Analysis tool to discover logical statements that will always evaluate to the same result. This is accomplished by remembering all the possible values or ranges of the variables being evaluated in the statement then building a logical table. If all statements resolve to the same outcome an error is reported

Miscellaneous Security Checking

Many security checks can be performed before compile time. Some system API functions are dangerous and should be only used with proper error checking, other functions are dangerous and should never be used. Possible buffer overrun conditions can be predicted so the developer may be able to fix them before they are discovered and exploited through the application. Time of check – Time of use problems can be discovered at build time as well, these errors occur when the application assumes that a resource has not been changed between the time it was checked and the time it is used. Other functions open the application to unnecessary security risks and can be mitigated by replacing these functions with similar, but more secure functions. This type of analysis is primarily useful for native code.

Metrics

Metrics can help a developer understand where there is unnecessary complexity in their code or metrics can generate helpful statistics for the application. Functional file coupling reports the relationships between the files of the application – uses and used by – and a metric that sums them both. Functional file cohesion shows other metrics within the file such as lines of code, number of methods, level of inheritance, and many others. Class level metrics allow the developer to gain an understanding of the cohesion of the application at a class level view. Cyclomatic complexity shows how many independent paths through each module the application can take; more paths can mean more complexity. Other useful metrics such as the ratio of comments to functional source code can point to places in the code that should be commented more effectively.

Simulator

The simulator is at once the most powerful and the most highly guarded part of the static analysis tool. The basic premise is as follows: the simulator selects a function and generates data based on each of the variables. If information exists about the possible data ranges of the variable then those ranges are used, however if no constructs are found for the data type then max and min are used as limits for the generated data.

Once the data has been generated the function is followed through each code path to hit each line of code. Function calls within the simulation can either be followed to their declaration, thus simulating another function or the return value of the function call can be generated for testing purposes.

This allows the simulator to test small sections of the code without requiring it to be compiled into the final executable application. Testing code this way can helps to alleviate some of the

shortcomings of traditional static analysis tools; however, it is very dependent on the ability to simulate 'interesting' data from a security point of view.

Crawl source code

These tools can crawl the source code of an application mapping out each possible code path, discovering unused or unreachable code. By mapping the possible values of a logical statement the analyzer can determine if the statement will have a constant outcome. Other difficult to determine problems with peer reviews can be quickly analyzed through sample values and logic tables; this can drastically speed up white box testing time. Unused or unreachable (dead) code poses a possible security risk to the application. Dead code is orphaned and unreachable so it remains untested throughout the product cycle. In a future release, a bug fix or code modification may allow the dead code to become active, thereby exposing untested security vulnerabilities or other functional flaws. Under certain extenuating circumstances a skilled hacker may be able to circumvent current constructs within the application and execute the untested code which may contain exploitable security flaws.

Limitations

Do not expect the scan to do more than scratch the surface. Even the best scanners have contextual problems. They are good at finding bugs that are caused by single lines of code, ok at finding bugs that span multiple lines of code in a single function, and generally bad at finding bugs whose scope spans multiple functions.

Managed code takes care of many of the bugs that scanners have been good at finding. In native code you could scan reasonably accurately for buffer overruns, format string problems, use of dangerous win32 APIs, memory leaks, etc. While some bugs types have been closed off, there are still numerous problems that can be found in managed code such as: Scoping problems, integer overflows, lack of cloning, exception handling, data truncation, lack of null checks and unchecked values used for memory allocation or buffer access.

STEP 3: CODE REVIEW PASS #2 – LOOK FOR COMMON BUGS

This step represents the 'meat' of the review process. In this step you will take a detailed look at the code with the goal of finding as many security vulnerabilities as possible.

You should use the set of goals you developed in step one for guidance. You should also have the following handy:

- List of hotspots from step 2
- An expert in the code you are looking at (this could be you)
- Usage scenarios
- Inputs and outputs
- Dataflow
- Areas that use native code
- Reference material

To conduct this code review pass you will:

- Augment the list of hotspots
- Step through the code looking for common bugs
- Perform dataflow analysis

Augment the Hotspots

If you have a static analysis tool and conducted a first code review pass with it you will have a list of hotspots that represent areas of code that were deemed particularly buggy by the tool. Now add to this list by conducting a sweep of the code looking for the following:

- **Inputs.** Review the list of inputs and then match this up to code you need to review. For instance you should flag public interfaces, UI, database interaction, socket interaction, file IO, pipes, and all other areas from which your component can accept data as critical for review.
- **Error handling code.** Especially look for areas with lots of dense error handling or very sparse error handling.
- **Complex code.** Look for areas of the code that don't seem easy to understand at first glance, you'll want to come back to these.
- **Routines that use cryptography.** There is guaranteed to be sensitive information being processed, you'll want to check these closely.
- **Use of CAS.** You'll want to review both declarative and imperative use of code access security.
- **Hard-coded strings or other values.** You'll want to see if sensitive data is revealed by hard-coding.
- **Commenting.** Flag areas of especially dense commenting, this is a hint of overly complex code. Also look at areas with especially sparse commenting, this is a sign of sloppy coding practices.
- **Interop.** Flag areas that call into native code, you'll want to look closely at each call.

You want this list to represent the code that has the highest priority for immediate review. If you run out of time you want to leave less important code un-reviewed.

Once you have a list of the critical areas you can begin the formal review and start the process of finding bugs.

Step through the Code Looking for Common Bugs

In this step you will take a detailed look at the code armed with the critical list of areas to review and the objectives for the code review.

By looking at the most important code first and employing a question-driven approach to finding bugs you can maximize your ability to find security vulnerabilities. This section contains a set of questions that you can start with. Over time, and multiple reviews, you can add questions to this list that are specific to your application so that what you learn can be applied to future reviews.

Two techniques for stepping through code during a review are described. Using a combination of both will result in the best review coverage.

- Control flow Analysis
- Dataflow Analysis

Control Flow Analysis

Control flow analysis is the mechanism used to step through logical conditions in the code. The process works as follows:

1. Look at a function and determine each branch condition. These can include loops, if statements, and try/catch blocks.
2. Understand the conditions under which each block will be executed.
3. Move to the next function and repeat.

As you investigate the control flow keep the following questions in mind.

1. Does the application rely on client side validation?

Managed or web app code is easy to modify on the client, the server should never trust client code. It is easy to modify the behavior of the client or just write a new client from scratch that doesn't observe the same data validation rules.

EXAMPLE

```
<html><head>
<script language='javascript'>
function validateAndSubmit(form)
{
  if(form.elments["path"].value.length() > 0)
  {
   form.submit();
  }
}
</script>
<form action="mypage.asp" method="post">
<input type='text' id='path'/>
<input type='button'
onclick='validateAndSubmit(this.parent)'>Submit</input>
</form>
</script></head><body>…</body></html>
```

In this example, client side scripting validates that the length of the "path" is greater than zero. If the server processing of this value relies on this assumption to mitigate a security threat, the attacker will have an easy time breaking the system.

2. Are there secrets or critical IP embedded in the code?

Managed code is easy to decompile, in fact it is possible to recover code from the final executable that is very similar to the original code. Any sensitive IP or hard coded secrets can be stolen with ease. An obfuscator can make this type of theft more difficult but cannot completely prevent it. Another common problem is to use hidden form fields thinking this information will not be visible to the user.

EXAMPLE

```
IntPtr tokenHandle = new IntPtr(0);
      IntPtr dupeTokenHandle = new IntPtr(0);
      string userName = "joe", domainName = "acmecorp",
      password="p@SswOrd";
      const int LOGON32_PROVIDER_DEFAULT = 0;
      //This parameter causes LogonUser to create a primary token.
      const int LOGON32_LOGON_INTERACTIVE = 2;
      const int SecurityImpersonation = 2;
      tokenHandle = IntPtr.Zero;
      dupeTokenHandle = IntPtr.Zero;
      // Call LogonUser to obtain a handle to an access token.
      bool returnValue = LogonUser(userName, domainName, password,
      LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT,
      ref tokenHandle);
```

3. Is sensitive data being stored in predictable locations (such as temp files), or being sent in clear text over the network?

Sensitive data should be stored and transmitted in encrypted form, anything less invites theft.

EXAMPLE

```
<!-- web.config -->
      <connectionStrings>
       <add name="MySQLServer"
```

```
    connectionString="Initial Catalog=finance;data
source=localhost;Integrated Security=SSPI;"
providerName="System.Data.SqlClient"/>
 </connectionStrings>
aspnet_regiis -pe "connectionStrings" -app "/MachineDPAPI" -prov
"DataProtectionConfigurationProvider"
```

A common error is to store server password in the ASP.NET web.config file:

Instead, the connection strings should be encrypted:

```
<!—web.config after encrypting the connection strings section -->
. . .
 <protectedData>
 <protectedDataSections>
  <add name="connectionStrings"
    provider="dataprotectionconfigurationprovider"
    inheritedByChildren="false" />
 </protectedDataSections>
 </protectedData>
 . . .
 <connectionStrings>
 <EncryptedData>
  <CipherData>
<CipherValue>AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAexuIJ/8oFE+sGTs7jBKZdgQAAAAC
AAAAAAADZgAAqAAAABAAAAAKms84dyaCPAeaSC1dIMIBAAAAAASAAACgAAAAEAAAAKaVI
6aAOFdqhdc6w1Er3HMwAAAAcZ00MZOz1dI7kYRvkMIn/BmfrvoHNUwz6H9rcxJ6Ow41E3h
wHLbh79IUWiiNp0VqFAAAAF2sXCdb3fcKkgnagkHkILqteTXh</CipherValue>
  </CipherData>
 </EncryptedData>
 </connectionStrings>
 . . .
<!—web.config after encrypting the connection strings section -->
```

Similarly forms authentication credentials should not be stored in the web.config file:

```
<authentication mode="Forms">
  <forms name="App" loginUrl="/login.aspx">
   <credentials passwordFormat = "Clear"
     <user name="UserName1" password="Password1"/>
     <user name="UserName2" password="Password2"/>
     <user name="UserName3" password="Password3"/>
   </credentials>
  </forms>
```

Instead use an external well ACLed user store.

4. Is cryptography being used?

Look for poor random number generators. You should ensure System.Security.Cryptography.RNGCryptoServiceProvider is being used to generate cryptographically secure random numbers.

Look for poor management of keys. Flag hard coded key values, leaving these in the code will help to ensure your cryptography is broken. Ensure that key values are not passed from method to method by-value as this will leave many copies of the secret in memory to be discovered by an attacker.

Look for failure to clear secrets from memory after use. Due to the fact that the CLR manages memory for you this is actually harder to do in managed code than it used to be in native code. In order to ensure secrets are adequately cleared make sure the following steps have been taken:

- Strings should not be used to store secrets; they are immutable and cannot be effectively cleared. Instead check to see that a byte Array or a CLR 2.0 SecureString has been used.
- Whichever type you use, make sure to call the .clear method as soon as you are done with the data.
- If your secret is paged to disk it can persist for long periods of time and be difficult to completely clear. Ensure that GCHandle.Alloc and GCHandleType.Pinned were used to keep the managed objects from being paged to disk.

Look for custom cryptographic routines. Ensure the use of System.Security.Cryptography. Cryptography is notoriously tricky to get right. The Windows crypto APIs are provably good, being implementation of algorithms derived from years of academic research and study. Some think that a less well known algorithm equals more security but this is not true. Cryptographic algorithms are mathematically proven, and as such the more eyes on them the better; obscurity will not protect your flawed implementation from a determined attacker.

5. Is CAS being used?

Look closely at each use of LinkDemand and Assert calls. These can open the code to luring attacks since the code access stack walk will be stopped before it is complete. While their use is sometimes necessary for performance reasons, make sure that there can be no un-trusted callers higher in the stack that could use this method's LinkDemand or Assert call as a mechanism for attack. Pay particular attention if the component you are reviewing allows partially trusted callers:

```
[assembly: AllowPartiallyTrustedCallersAttribute()]
```

This will allow the assembly to be accessible from callers which are not fully trusted. If the component you are reviewing then calls into an assembly that does not allow partial trusted callers a security bug could result.

Also check for requests for dangerous permissions such as: UnmanagedCode, MemberAccess, SerializationFormatter, SkipVerification, ControlEvidence / ControlPolicy, ControlAppDomain, ControlDomainPolicy, SuppressUnmanagedCodeSecurityAttribute.

EXAMPLE

```
[DllImport("Crypt32.dll", SetLastError=true,
CharSet=System.Runtime.InteropServices.CharSet.Auto)]
[SuppressUnmanagedCodeSecurity]
private static extern bool CryptProtectData(
      ref DATA_BLOB pDataIn,
      String szDataDescr,
      ref DATA_BLOB pOptionalEntropy,
      IntPtr pvReserved,
      ref CRYPTPROTECT_PROMPTSTRUCT pPromptStruct,
      int dwFlags,
      ref DATA_BLOB pDataOut);
```

6. Are there undocumented public interfaces?

Look for public interfaces that have not been documented as well as the other interfaces in your application. Is the interface you are looking at a test interface or a backdoor administrative interface? Many times these shouldn't be in the product at all, and they are almost never given the same level of design and test scrutiny as the rest of the product.

7. Is the component giving dependencies too much trust?

Without explicit safeguards it is possible for an attacker to trick your code into loading a malicious library instead of trusted code. Check to see if all the loaded assemblies are strongly named; this

step ensures that tampering cannot occur. Without strong names your code could be calling into malicious code without knowing it. The use of native code libraries makes this harder to do so be cautious about trusting native code implicitly. Native libraries can be checked with a hash or a certificate. Additionally you should make sure that all libraries are loaded with a complete path in order to avoid canonicalization attacks.

Also check whether delay signing is enabled. Delay signing is generally regarded as a good practice since it helps protect the confidentiality of the private key that will be used for signing the component:

```
[assembly:AssemblyDelaySignAttribute(true)]
```

8. Is there proper and consistent error checking?

Ensure that there is consistent use of try/catch and return value checking. Keep an eye out for empty catch blocks. Double check error handling every time an assembly is loaded dynamically; look for calls to System.Reflection.Assembly.Load. Make sure that if a library contains security functionality and it fails to load that the code defaults to higher security.

Also look for cases where impersonation or elevated privileges may not be lowered in the case an exception is thrown. This can occur either due to a logic bug – catch doesn't contain the right code – or due to a subtle misuse of a finally block by an attacker.

Exception filters run before the finally block so it may result in malicious code executing in the context of the privileged code rather than in the partially trusted context it should be running in.

EXAMPLE

Bad logic -

```
try
{
      ElevatePrivilege();
      //If ReadSecretFile throws an exception privileges will not be
      lowered
      ReadSecretFile();
      LowerPrivilege();
}
catch(FileException fe)
{
      ReportException();
}
```

Misuse of finally –

```
'Malicious VB Client
Imports VictimLib
Module MalCode
  Sub Main()
    Dim victim As New Victim
    Try
      victim.Operation(-1)
    Catch When Malware(victim) = True
    End Try
  End Sub
  Function Malware(ByVal victim As Victim) As Boolean
      'Do malicious stuff in here
    Return True
  End Function
End Module
'Victim Server
using System;
namespace VictimLib {
  public class Victim {
    public void Operation(int param) {
  try {
            RaisePrivilege();
            if(param < 0) {
                  throw new ArithmeticException("Invalid input");
            }
      }
        /*
        catch(Exception e_) {
            LowerPrivilege();
            throw(e_);
        }
        */
      finally {
            LowerPrivilege();
      }
    }
  }
}
```

9. Do error messages give away too much information?

Error messages should be helpful to the average user without giving away information that an attacker could use against you. Ensure that the code doesn't give away call stacks, lines of code, server file paths or anything else internal to the application. This information is not helpful to a user but can be very helpful to an attacker.

Make sure custom error pages have been implemented in ASP.NET applications in order to ensure no sensitive data is given away and make sure application tracing has been turned off.

Check security sensitive error paths with caution. For instance, be careful about changing error messages for differing error code paths during user authentication. A common problem is to display different error message for bad username/bad password vs. good username/bad password. While the difference in errors may be subtle the end result will give the attacker information that they can use against you.

Example:

Logging in with a bad username gives one error message.

Logging in with a good username but a bad password gives a different error message.

10. Does your application expose sensitive information via user session?

If you are reviewing a web application and it reveals sensitive information via user session then pay particular attention to how the session is managed. In ASP.Net the session ID is properly randomized so it is hard to guess session IDs, however, there are other ways an attacker can get at this information. Make sure that the session ID is sent over SSL, and check to ensure that the session timeout is short:

> <sessionState mode="InProc" stateConnectionString="tcpip=127.0.0.1:42424" sqlConnectionString="data source=127.0.0.1;user id=<username>;password=<strong password>" cookieless="false" timeout="20" regenerateExpiredSessionId = “true” />

11. Can write operations be performed with a GET request?

Check to see if it is possible to modify data or content with a GET request. This opens the door for an attacker to trick a legitimate user into performing illegitimate actions with their account. In an ASP.Net application ensure that Request.RequestType is checked before making a SQL query or any other operation that can cause data or content changes.

12. Is the code multithreaded?

Check for race conditions, especially in static methods and constructors.

EXAMPLE

```
private static int amtRecvd = 0;
public static int IncrementAmountReceived(int increment)
```

```
{
        return(amtRecvd += increment);
}
```

If two threads call into this code at once it could result in an incorrect calculation on the amtRecvd value.

Dataflow analysis

The previous questions will help you find a large number of bugs, next you will want to use a technique called dataflow analysis to find bugs associated with poor input handling. Dataflow analysis is the mechanism used to trace data from the points of input to the points of output. Since there can be many data flows in your application, use the prioritization list you built earlier to focus your work. The process works as follows:

1. For each input location determine how much you trust the source of input. When in doubt you should give it no trust.
2. Trace the flow of data to each possible output noting along the way any attempts at data validation.
3. Move to the next input and continue.

When you are done you should have a list of all the functions that each piece of input data touches as well as the eventual outputs where it ends up. Don't forget to pay attention to areas where the data is parsed and may end up in multiple output locations. Also pay attention to intermediary output locations. For instance the input may end up in a database and then later placed in web page content.

Thinking about how much you trust each input source is tricky. Ideally you will trust no input that comes from outside your component and validate all data fully. For performance and maintainability reasons, however, this may not always be practical. In general you can trust code that is closest to you and give less trust to code that is less well known. Here is an example of how to think about trust boundaries:

- High trust
    - Input that comes from code you are reviewing inside the component
    - Input that comes from known-good strongly named managed assemblies or signed/hashed native libraries
    - Input that comes from a database that is only used by your component and for which you can prove that all data to the database has been properly validated
    - Network data that has been signed by a known good source (IPSec or SSL)
- Medium trust
    - Input that comes from known-good assemblies or native libraries that have not been strongly named or signed but are local to your server

    - Input from that comes from a public interface that should only be accessible to trusted users
    - Input from that comes from a UI that should only be accessible to trusted users
    - Network data that should not be accessible to an untrusted user (such as a segmented LAN internal to your datacenter)
- Low trust
    - Input that comes from assemblies or native libraries that have not been strongly named or signed and are located on the client
    - Input that comes from client code
    - Input that comes over the network
    - Input that comes from a file
    - Input that comes from a public interface that is accessible to any user
    - Input that comes from UI that is accessible to any user
    - Input that comes from a database that is shared with other components or processes

Remember to trace all the way to the source and assign trust based on the weakest link.

As you conduct your traces look at the code carefully to ensure that input validation is performed rigorously on low trust input and performed adequately on medium trust input. Ideally you will have a set of common validation routines that can be called into as soon as un-trusted data is received by the application. This gives a central point of failure that can be updated as new information is discovered. However, in addition to knowing how much you trust the data you must also be aware of how the data is going to be used in order to know how it should be validated. This is where the dataflow analysis becomes important. For instance if the eventual output for the untrusted data is a database then you should check for SQL injection problems. If the data will be used to make a calculation then you should check for numeric overflows and underflows. If the data will be displayed in a web page then you should check for cross site scripting problems. Keep in mind while reviewing validation routines that validation should always opt-in, not opt-out; it's easier to accurately define what's good rather than what's bad.

The most common input validation bugs are as follows:

Numeric underflows and overflows

This problem is caused when a calculation causes a data value to be larger or smaller than its data type allows. This will cause the value to wrap-around and generally become much larger or smaller than expected. For instance assigning -1 to an unsigned integer will result in the actual value becoming larger than four billion. VB will throw an exception when as soon as an overflow or underflow occurs, however, C# will not. As you are tracing data through the code ensure that any location where a user can give input that results in a calculation will not cause an underflow or overflow condition.

EXAMPLE

```
int[] filter(uint len, int[] numbers)
{
      uint newLen = len * 3/4;
      int[] buf = new int[newLen];
      int j = 0;
      for(int i = 0; i < len; i++)
      {
          if (i % 4 != 0)
          buf[j++] = numbers[i];
      }
      return buf;
}
```

Problem is that in calculating the value for len, the code first computes len * 3 and then divides by 4. When len is large enough (~1.4 billion), len * 3 overflows and newLen gets assigned too small of a value. The result in this code will be an unhandled IndexOutOfRange exception.

SQL injection

A SQL injection attack occurs when un-trusted input can modify the logic of a SQL query in unexpected ways. As you are tracing through the code ensure that any input that is used in a SQL query is validated or make sure that the SQL queries are parameterized.

Example

The SQL query in code looks like this.

```
query = "SELECT * FROM USERS WHERE USER_ID = '" + userIdFromWebPage + "'";
```

userIdFromWebPage is a variable that contains untrusted data that has not been validated. Imagine that it contains '" or 1=1 –", or '" ;DROP TABLE users –", or '" ;exec xp_cmdshell('format c:') –". The final query could look like this.

```
"select * FROM USERS WHERE USER_ID = '' ;exec xp_cmdshell('format c:') –"
```

This will result in a format of the c:\ drive on the database server.

The code should look like:

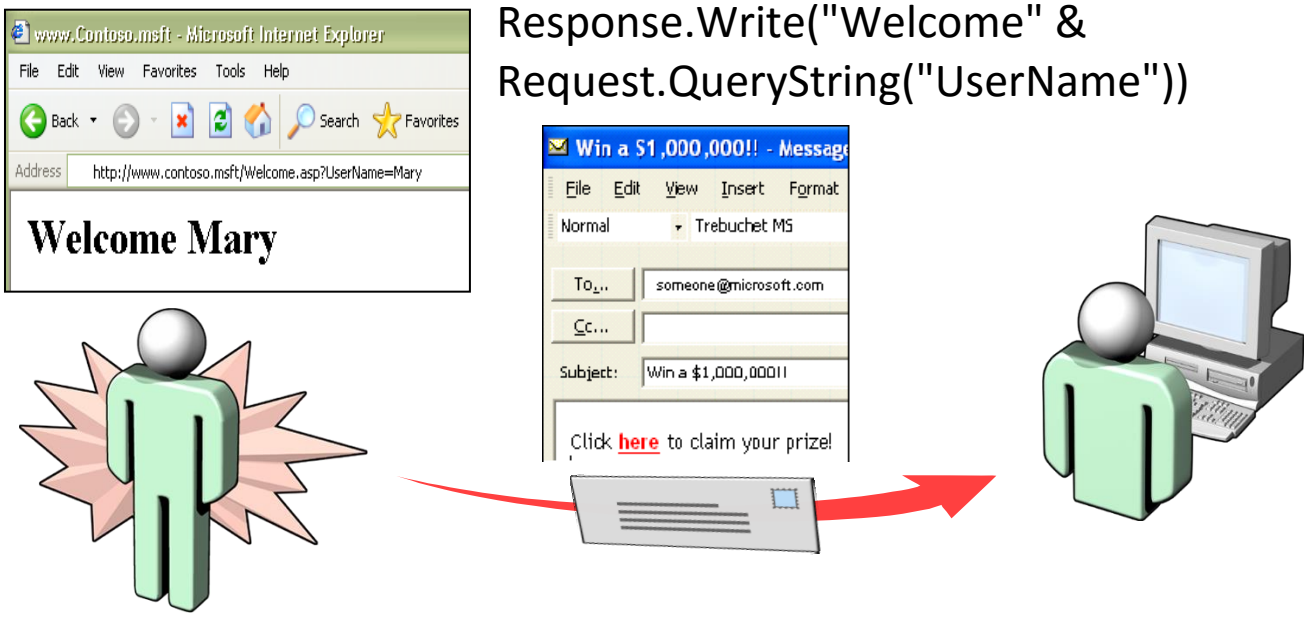
```
SqlCommand queryCMD = new SqlCommand("GetUser", sqlConn);
queryCMD.CommandType = CommandType.StoredProcedure;

SqlParameter myParm = queryCMD.Parameters.Add("@UserID",
SqlDbType.Int, 4);
myParm.Value = userIdFromWebPage;

SqlDataReader myReader = queryCMD.ExecuteReader();
```

Cross Site Scripting

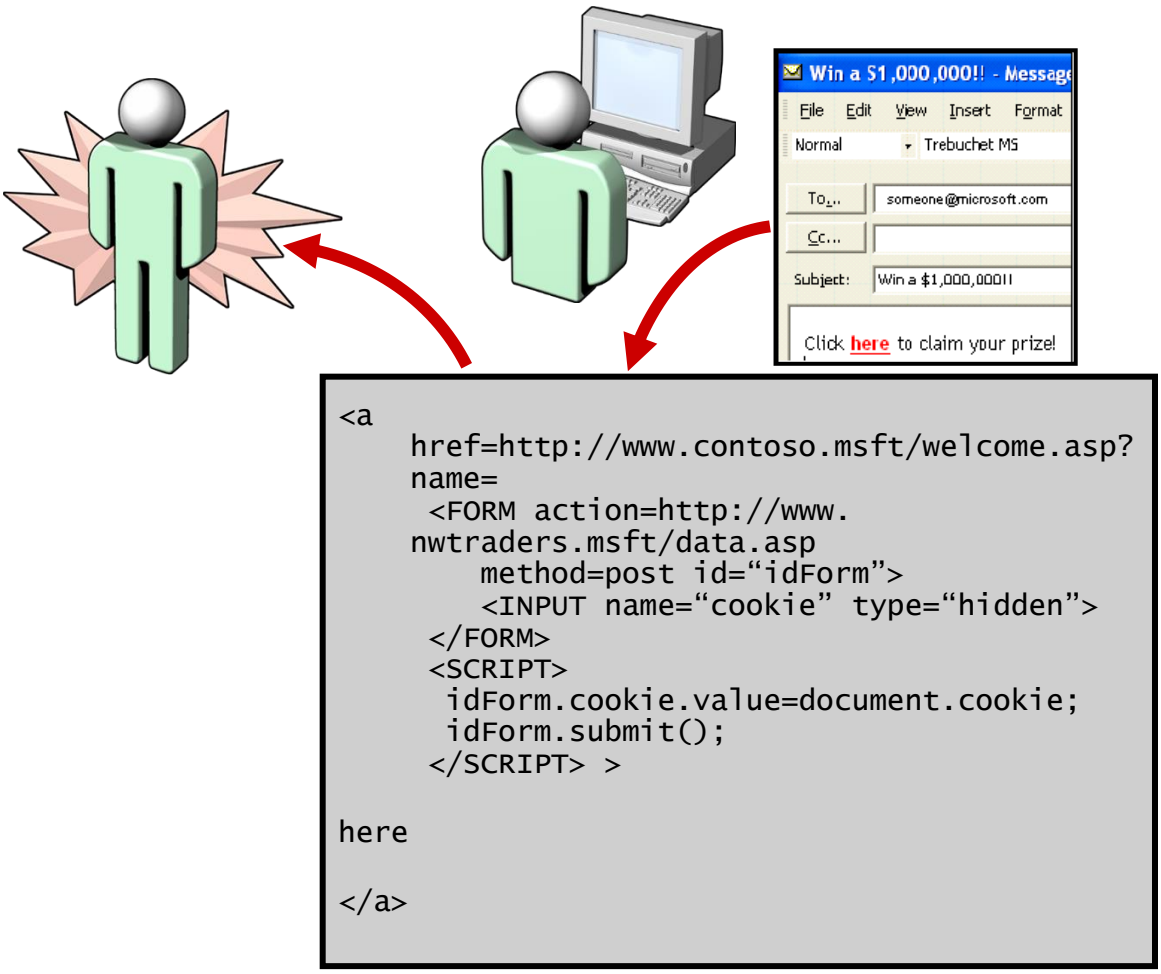
Cross-site scripting is caused when an attacker manages to input script code into an application so that it is echoed back and executed in the security context of the application. This allows an attacker to steal user information including forms data and cookies. This vulnerability may be present whenever a web application echoes unfiltered user input back out to web content.

As you are tracing through the code ensure that un-trusted data whose ultimate output is web page content does not contain HTML tags. Be aware that the data could move from un-trusted input to web page output via a roundabout path – for instance through a database and then later queried out of the database and displayed on a web page. To protect against this bug, ensure HTMLEncode or URLEncode are used before user input is echoed back to web content.

Example



In this example, the web application is hosted at http://www.contoso.msft. The ASP script is designed to output “Welcome <UserName>”. This represents a cross-site scripting vulnerability because the value of UserName is not filtered or encoded. The attack proceeds as follows: 1.) An attacker sends email to an unsuspecting user containing malicious HTML form, 2.) When the user clicks on the link in the email, a post request is sent to the Contoso page. The request contains JavaScript for the value of the username field, 3.) The Contoso site sends the value of “username”, really the JavaScript generated by the attacker, to the user’s browser and the browser executes thinking the source of the script is the Contoso site. In this case, the script sends any cookie data associated with the page to the attacker’s machine.



Canonicalization

Canonicalization errors occur whenever there are multiple ways to represent a resource *and* the different representations result in varying security logic being run. There are a several resource types for which this problem can occur:

- File resources
  - Use of partial paths may result in a file other than what you expect being loaded.
  - Use of the PATH environment variable may give control of the paths used by your application to an attacker.
- URLs
  - Alternate representation of an IP address such as dotless IP may result in an URL other than what you expected being loaded.
  - Encoded characters such as %20 for space may result in an URL other than what you expected being loaded.

The result of this bug is that an attacker gains access to a resource they would not otherwise have access to. As you are tracing through the code look carefully at areas where resources are being accessed based upon user input. Make sure that file names are canonicalized before use with Path.GetFullPath. Make sure that URLs are canonicalized before use with Uri.AbsoluteUri.

Use CAS for an extra layer of protection. Refuse permissions that are not needed as well as indicate to the runtime what you need.

EXAMPLE

```
[assembly:FileIOPermission( SecurityAction.RequestMinimum, Read =
"c:\\temp" )]
[assembly:FileDialogPermission( SecurityAction.RequestOptional )]
[assembly:FileIOPermission( SecurityAction.Deny, Write =
"c:\\windows" )]
```

Native code issues

If the code you are reviewing makes calls into native code there are some additional checks that must be done to ensure security. Assumptions made in managed code about data validity and memory soundness may not hold true when running native code. The following bugs that are impossible or unusual in managed code are quite possible in native code.

Buffer overflows

Buffer overruns are a classic vulnerability that may lead to execution of arbitrary code. Successful exploitation of this vulnerability relies on low level details of the system architecture that is outside the scope of this guide, however, given the magnitude and frequent occurrence of the problem it is important to recognize when these vulnerabilities are present.

Buffer Overflows are largely prevented by managed code since the common language runtime abstracts out the underlying machine architecture. However, any code marked unsafe is allowed direct memory access and therefore can contain buffer overflows. Also, many applications contain a mixture of managed and unmanaged code and so the unmanaged code must be looked at closely.

While tracing through the code ensure that for unsafe code the following rules are followed:

Make sure any functions that copy variable length data into a buffer take and use a maximum length parameter properly.

- Never rely on another layer or tier for data truncation.
- If you see a problem always choose to truncate data instead of expanding the buffer to fit. Buffer expansion may just move the problem downstream.
- Ensure any unmanaged code was compiled with /GS option.

EXAMPLE

```
public void ProcessInput
{
  char[] data = new char[255];
  GetData(data);
}
public unsafe void GetData(char[] buffer)
{
  int ch = 0;
  fixed (char* pBuf = buffer)
  {
      do
      {
            ch = System.Console.Read();
            *(pBuf++) = (char)ch;
      }while(ch != '\n');
  }
}
```

An overflow occurs whenever a single line is more than 255 characters long. There are two problems in this code:

1. The ProcessInput function only allocates enough space for 255 characters.
2. The GetData function does not check the size of the array as it fills it.

Format string

Format String bugs emanate from the printf family of functions handling of variables, and the %n format directive. The printf family of functions will pop the stack as many times as they see "%" in the format string. Sufficient %'s can traverse the stack, and reach any location in it, or above. Additionally the use of %n can allow arbitrary writing of data anywhere within the stack.

While tracing through the code make sure that format string data never contains user input. As this can only occur in unmanaged code it is only worth worrying about if untrusted input is used in a call to a native library.

EXAMPLE

```
void main (int argc,
          char **argv)
{
  /* Whatever the user said, spit back! */
  printf (argv[1]);
}
```

In this example untrusted input in the form of a command line parameter, is passed directly to a printf statement. This means an attacker could include format string % directives into the string and force the application to return or modify arbitrary memory on the stack.

STEP 4: CODE REVIEW PASS #3 – LOOK FOR BUGS UNIQUE TO THIS APPLICATION ARCHITECTURE

In this step you will look at your list of code review objectives and cover anything that has not yet been covered. It is likely that there are potential bugs based on a unique security architecture that has been implemented or for threats that were recognized in the threat model and already mitigated. The final code review pass is focused on verifying implementation of these security features that are unique to your application architecture. Just as was done in the previous steps, employing a question-driven approach will produce the best results:

1. Is there a security architecture implemented in this application?

A custom security architecture is a great location to look for bugs for several reasons:

1. It has already been recognized that a security problem exists, that's why the custom security code was written in the first place.
2. Unlike other areas of the product, a functional bug is very likely to result in a security vulnerability.

2. Are there unique roles in the application?

The use of roles assumes that there are some users with lower privileges than others. Ensure there are no problems in the code that could allow one role to assume the privileges of another.

The first step is to understand what the set of roles and what each role should be allowed to do. This can be accomplished with a simple matrix that contains privileges in rows and roles in columns. Make a checkmark in each cell that corresponds to a privilege allowed by a role.

Once the matrix has been completed review the code for contradictions to this matrix. Even a well designed system with clearly drawn out roles can be broken by a bad assumption or a logical mistake in the implementation.

Example

	Objects				
Subjects	User creation	Permission modification	Object creation	Object removal	Object read
Admin	x	x			
Content Creator			x	x	x
Reader					x
Anonymous					x

TOOL ROUNDUP

While static analysis tools are quite helpful during a code review, there are others that can be useful as well. Here is a roundup of the tool types:

Static analysis tools

- Pros
  - Can find logical and security errors
- Cons
  - Finds primarily simple bugs
  - False positives
  - False sense of security

Dynamic analysis tools

- Memory profilers, fault injection tools, and code coverage profilers
- Pros
  - Good at finding more complex errors that only surface at runtime
- Cons
  - Memory profilers are only useful in unmanaged code
  - Only work on the code path that has been executed (hard to get good coverage)
  - Don't find that many security problems, through they can help with general reliability

Fuzzers

- Used to provide bad input to your application
- Pros
    - Can find surprising combinations of input that you may have missed in your code review
    - Usually not much work to implement
    - Random data can be surprisingly effective at finding bugs
- Cons
    - Not good at finding stateful bugs
    - Best suited for file or network corruption – also can be applied to interfaces but with less success

Vulnerability scanners

- Pros
    - Can help to ensure that a web application is not vulnerable to known attack vectors
    - Can find a wide variety of security vulnerabilities such as server configuration errors, known vulnerable plug-ins and extensions, SQL injection, cross site scripting
- Cons
    - Like static analysis tools they only find low hanging fruit
    - They have trouble getting full coverage – sometimes befuddled by authentication, have trouble with stateful applications
    - Scans can be time consuming – both the scan and the analysis
    - False positives
    - False sense of security

NEXT STEPS: POST CODE REVIEW ACTIVITIES

After you are done with your code review, do the following:

- **Prioritize the bugs found –** prioritization should be based upon the impact the bug will have on your customers. Think through the maximum damage potential as well as which of your customers will be impacted.
- **Fix the right set of bugs –** each bug fixed can introduce a new unknown bug. Sometimes the bug you know is less dangerous than the bug you don't
- **Learn from your mistakes –** keep a running dialog within your team discussing the mistakes made, how they were found, how they were fixed. Strive to write code that comes up clean in a code review the first time!

About Security Innovation

Security Innovation is an established leader in the software security and cryptography space. For over a decade, the company has provided products, training, and software assessment services to help organizations build and deploy more secure software systems and harden their data communications schemes.

Security Innovation built upon its core competencies in application security with the acquisition of NTRU CryptoSystems in 2009, a company that developed proprietary, standardized algorithms. This resulted in the strongest and fastest public key cryptography available - and the means to overcome historical implementation and speed barriers that have plagued the data encryption industry. With these core strengths intact, Security Innovation is in a position to help organizations protect their data at two critical points: while applications are accessing it and during transmission. The company's flagship products include TeamProfessor, the industry's largest library of software security eLearning titles, and TeamMentor, the industry's first secure coding standards product.

Security Innovation is privately held and is headquartered in Wilmington, MA USA. More information can be found at www.securityinnovation.com.