

Do the changes made in ASP .NET, from version 1.X to 2.0, improve the security?

Written by Johan Gille

Department of Computer and Systems Sciences
Royal Institute of Technology

This thesis corresponds to 20 weeks of full – time work.

Abstract

The problem, from a security perspective, with every Web application is that they are made available to the public via the World Wide Web (WWW). The biggest challenge for a Web platform development company is to make security countermeasures as easy as possible for Web application developers to implement. This is exactly what Microsoft have tried to achieve with its new release of Microsoft .NET (2.0).

The thesis is created to be as good and simple as possible. The basic idea is to divide all the most common Web application vulnerabilities into categories and then systematically tackle them one by one. By doing this you get a really good overview of each category's main threats and its ASP .NET countermeasure in both version 1.X and 2.0.

To summarize the changes, made from ASP .NET 1.X to ASP .NET 2.0, you can say that they have "patched" most of the shortcomings in ASP .NET 1.X, such as the HttpOnly cookie attribute, AES encryption is now available for forms authentication and it's now possible to encrypt parts of .config files. The biggest change though is the prebuilt features, such as auditing and logging, authentication and authorization. The reason why they are prebuilt is to increase productivity, which in turn, indirectly, increases security. The idea is very good from a security perspective, if you intend to start from scratch and have no intention of implementing anything by your self, but if you already have a database you've to rewrite some of the chosen provider's code or create a provider of your own.

The only thing that I really miss, from a security perspective, is a default implementation of authentication ticket / cookie logging, which is a shame since it makes cookie replay attacks possible even in the new version. But the overall judgment of ASP .NET 2.0's security countermeasure must be positive although the improvements doesn't dramatically improve the security level.

Table of contents

1 Introduction	4
1.1 Background	4
1.2 Problem.....	4
1.3 Question.....	4
1.4 Objective.....	4
1.5 Method.....	4
1.6 Limitation	4
1.7 Audience	5
2 ASP .NET security	6
2.1 How to secure an ASP .NET Web application	6
2.2 Distributed ASP .NET Web application	6
2.2.1 Logical deployment	7
2.2.2 Physical deployment.....	7
2.2.2.1 The Web server as an application server deployment pattern.....	7
2.2.2.2 The remote application tier deployment pattern	7
2.3 Vulnerability categories	7
2.3.1 Auditing and logging.....	8
2.3.1.1 Threat: Attackers cover their tracks.....	8
2.3.1.2 Threat: Attackers exploit an application without leaving a trace	8
2.3.1.3Threat: User denies performing an operation	9
2.3.1.4 Design guidelines for auditing and logging attacks.....	9
2.3.1.5 Auditing and logging countermeasures in ASP .NET 1.X and 2.0	9
2.3.1.5 Auditing and logging countermeasure in ASP .NET 2.0	12
2.3.2 Authentication and authorization.....	22
2.3.2.1 Authentication threat: Brute force attacks	23
2.3.2.2 Authentication threat: Cookie replay attacks.....	23
2.3.2.3 Authentication threat: Credential theft	23
2.3.2.4 Authentication threat: Dictionary attacks.....	23
2.3.2.5 Authentication threat: Network eavesdropping.....	24
2.3.2.6 Authorization threat: Data tampering.....	24
2.3.2.7 Authorization threat: Disclosure of confidential data	24
2.3.2.8 Authorization threat: Elevation of privilege	25
2.3.2.9 Authorization threat: Luring attacks.....	25
2.3.2.10 Design guidelines for preventing authentication attacks.....	25
2.3.2.11 Authentication and authorization countermeasures in ASP .NET 1.X	26
2.3.2.12 Authentication and authorization countermeasures in ASP .NET 2.0	33
2.3.3 Cryptography	37
2.3.3.1 Threat: Checksum spoofing	37
2.3.3.2 Threat: Poor key generation or key management	38
2.3.3.3 Threat: Weak or custom encryption	38
2.3.3.4 Design guidelines for preventing cryptography attacks.....	38
2.3.3.5 Cryptography and sensitive data countermeasure in ASP .NET 1.X	39
2.3.3.6 Cryptography and sensitive data countermeasure in ASP .NET 2.0	47
2.3.4 Exception management	52
2.3.4.1 Threat: Attacker reveals implementation details.....	52
2.3.4.2 Threat: Denial of service (DOS)	52
2.3.4.3 Design guidelines for preventing exception management attacks....	53

2.3.4.4 Exception management countermeasures in ASP .NET	53
2.3.5 Input validation	54
2.3.5.1 Threat: Buffer overflows.....	55
2.3.5.2 Threat: Canonicalization.....	55
2.3.5.3 Threat: SQL injection.....	56
2.3.5.4 Threat: XSS.....	57
2.3.5.5 Design guidelines for preventing input validation attacks.....	57
2.3.5.6 How to implement these guidelines into practice	59
2.3.5.7 Input validation countermeasures in ASP .NET 1.X	60
2.3.5.8 .NET countermeasures for XSS attacks.....	64
2.3.5.9 Input validation countermeasures in ASP .NET 2.0.....	65
2.3.6 Parameter manipulation.....	67
2.3.6.1 Threat: Cookie manipulation	67
2.3.6.2 Threat: Form field manipulation	67
2.3.6.3 Threat: HTTP header manipulation	67
2.3.6.4 Threat: Query string manipulation.....	67
2.3.6.5 Design guidelines for preventing parameter manipulation attacks..	68
2.3.6.6 Parameter manipulation countermeasures in ASP .NET.....	68
2.3.7 Sensitive data	69
2.3.7.1 Threat: Access to sensitive data in storage	69
2.3.7.2 Threat: Data tampering.....	69
2.3.7.3 Threat: Network eavesdropping	70
2.3.7.4 Design guidelines for preventing Sensitive data attacks.....	70
2.3.7.5 Sensitive data countermeasures in ASP .NET	71
2.3.8 Session management	72
2.3.8.1 Threat: Man in the middle	72
2.3.8.2 Threat: Session hijacking	72
2.3.8.3 Threat: Session replay	72
2.3.8.4 Design guidelines for preventing session management attacks.....	73
2.3.8.5 Session Management countermeasures in ASP .NET	73
3 Summary and conclusion	76
4 Future study and improvements	77
5 Resources, Reference list	78

1 Introduction

1.1 Background

The pure and simple reason why I wrote this thesis is because it combines two of my biggest interests; Microsoft .NET and IS / IT security. Since Microsoft .NET is out of the scope of this thesis and that's why it's limited to a part of it; ASP .NET, which is Microsoft's new initiative of creating Web form based Web applications.

1.2 Problem

The problem, from a security perspective, with every Web application is that they are made available to the public via the World Wide Web (WWW). The biggest challenge for Web platform development companies is to make implementation of security countermeasures as easy as possible for Web application developers. This is exactly what Microsoft have tried to achieve with its new release of Microsoft .NET (2.0).

1.3 Question

The question I want answered, in this thesis, is whether Microsoft have managed to improve the security countermeasures in ASP .NET 2.0 in comparison to ASP .NET 1.X.

1.4 Objective

My objective with this thesis is to give the reader a good knowledge and understanding of the security countermeasures in Web form based ASP .NET 1.X and ASP .NET 2.0 Web applications, so that they later can develop secure ASP .NET Web applications.

1.5 Method

The obvious source of information for this kind of thesis is of course the Internet and books. I also did some testing of the countermeasures that I didn't fully understand. These tests were only done to help myself produce a better explanation of the countermeasure in the thesis.

The layout of this thesis is inspired by how a good MSDN Library book / Web site tackled the issue of presenting ASP .NET's security countermeasures in a good and simple way. The basic idea is to divide all the most common Web application vulnerabilities into categories and then systematically tackle them one by one. By doing this you get a really good overview of each category's main threats and its ASP .NET countermeasure in both ASP .NET 1.X and ASP .NET 2.0.

1.6 Limitation

Since Microsoft .NET 2.0 is out of the scope for a single thesis, I limited myself to a part of it; Web form based ASP .NET Web applications.

1.7 Audience

The audience of this thesis should have good programming skills, good understanding of Microsoft .NET or similar technologies and have a really good knowledge of IS / IT security.

2 ASP .NET security

ASP .NET security and security in general is fundamentally about protecting your assets. An asset can be everything from your Web page, Web site, Web application or customer database to your company's reputation.

What you're protecting your assets from is threats. A threat is any potential occurrence, malicious or otherwise, that could harm an asset. The possibility for harm to occur is called risk. Security is therefore all about risk management and implementing effective countermeasures.

The weakness that makes a threat possible is called a vulnerability. A vulnerability can be caused by poor design, configuration mistakes or inappropriate and insecure coding techniques. Weak input validation is one example of a vulnerability, which can result in input attacks. An attack is an action that exploits a vulnerability or enacts a threat.

2.1 How to secure an ASP .NET Web application

To create a secure and "hack resilient"¹ distributed ASP .NET Web application, you need to have a holistic and systematic approach when securing your Web application's hosts and network.

The core elements of a secure network are; the firewall, the router and the switch. For more information about these elements, read chapter 15 in *Improving Web application security: Threats and Countermeasures*.

The hosts are a little more complex since they include the operating system (OS), the platform components and services, such as the .NET Framework, Internet Information Services (IIS) and SQL Server 200X, and the runtime components and services, such as the ASP .NET runtime environment, which is the runtime environment that executes ASP .NET's web applications and web services.

This report will, from now on, mainly investigate the security related changes made to ASP .NET, from version 1.X to 2.0. Information on how to secure the rest of the hosts can be found in chapter 16 – 18 in *Improving Web application security: Threats and Countermeasures*.

2.2 Distributed ASP .NET Web application

The reasons why you deploy parts of a Web application into several different tiers are to make it more maintainable and scalable. Although this isn't a direct security question, it's extremely important to understand how a security minded and serious company would distribute their (Web) application(s).

A Web application can and often are deployed in two different ways; logical and physical.

¹ A hack resilient Web application is a Web application that reduces the likelihood of a successful attack and mitigates the risk if an attack occurs.

2.2.1 Logical deployment

The logical ASP .NET Web application architecture views any system as a set of tiers. The tiers, which normally are used, are the following:

- **The presentation tier.** The presentation tier is responsible for the client interaction with the system and to provide a bridge to the business tier. This logical tier is, in this report, represented by the native part of the .NET Framework named ASP .NET.
- **The business tier.** The business tiers provide the core functionality of the system. To access data in a database, this tier uses the data access tier.
- **The data access tier.** The data access tier acts as an interface to the data tier. This tier knows from which database to retrieve and store data. This logical tier is, in this report, represented by the native part of the .NET Framework named ADO .NET.
- **The data tier.** This tier is responsible for retrieving, storing and updating data. An example of a typical data tier is SQL Server.

2.2.2 Physical deployment

The number of logical tiers doesn't imply the number of physical tiers. All logical tiers may be physically located on the same server or spread across multiple servers. The two most common physical deployment patterns are presented below.

2.2.2.1 The Web server as an application server deployment pattern

The "Web server as an application server" deployment pattern locates the presentation, business and data access tiers on the Web tier (Web server) and the data tier on the database tier (database server). Although this deployment pattern isn't particularly scalable, it minimizes the network traffic, which improves the performance of your Web application.

2.2.2.2 The remote application tier deployment pattern

The "remote application tier" deployment pattern locates the presentation tier on the Web tier, the business and data access tiers on the remote application tier (application server) and the data tier on the database tier. This deployment pattern is very scalable, but it increases network traffic, which decreases the performance of your Web application. This deployment pattern is also particularly useful when the Web tier, by itself, needs to be contained within a perimeter network (also known as a demilitarized zone (DMZ)), which means that the Web tier is separated from both end users and the remote application tier with packet filtering firewalls.

2.3 Vulnerability categories

All Web applications have the same security vulnerabilities. By organizing these vulnerabilities into categories, you can, in a much easier way, systematically tackle them when designing and implementing Web applications. The categories used in this report are:

- **Auditing and logging.**
- **Authentication.**
- **Authorization.**
- **Configuration management.**
- **Cryptography.**
- **Exception management.**
- **Input validation.**

- **Parameter manipulation.**
- **Sensitive data.**
- **Session management.**

This report will tackle each category on its own and each category will include the following:

- **A brief explanation of itself.**
- **A presentation and explanation of its threats.**
- **A presentation of its design guidelines.**
- **A presentation and explanation of the countermeasures available in ASP .NET.**

2.3.1 Auditing and logging

The auditing and logging vulnerability category addresses the question: who did what and when?

Auditing is all about how you analyze your Web application's logs and data. Efficient auditing is a vital aid in identifying intruders or attacks in progress.

Logging is all about how your Web application logs security related events. Efficient logging proves particularly useful as forensic information when determining how an intrusion or attack was performed. It's much harder for a user to deny performing an operation if a series of synchronized log entries, on multiple servers, indicate that the user performed that operation.

Efficient auditing and logging is therefore the key to preventing threats, such as; foot printing, password cracking attempts and repudiation. The top auditing and logging related threats include:

- **Attackers cover their tracks**
- **Attackers exploit an application without leaving a trace**
- **User denies performing an operation**

2.3.1.1 Threat: Attackers cover their tracks

Your log files must be well protected to ensure that attackers are not able to cover their tracks.

Countermeasures to help prevent attackers from covering their tracks include:

- Secure log files by using restricted Access Control Lists (ACLs).
- Relocate system log files away from their default locations.

2.3.1.2 Threat: Attackers exploit an application without leaving a trace

System and application level auditing is required to ensure that suspicious activity doesn't go undetected.

Countermeasures to detect suspicious activity include:

- Log critical application level operations.
- Use platform level auditing to audit login and logout events, access to the file system and failed object access attempts.
- Back up log files and regularly analyze them for signs of suspicious activity.

2.3.1.3 Threat: User denies performing an operation

The issue of repudiation is concerned with a user denying that he or she performed an action or initiated a transaction. You need defense mechanisms in place to ensure that all user activity can be tracked and recorded.

Countermeasures to help prevent repudiation threats include:

- Audit and log activity on all servers.
- Log key events such as transactions and login and logout events.
- Do not use shared accounts since the original source cannot be determined.

2.3.1.4 Design guidelines for auditing and logging attacks

You should audit and log activity across all of your application's logical and physical tiers. With efficient logs and audits, you enormously improve the likelihood of detecting suspicious activity, which might be an early indication of a full blown attack.

The following auditing and logging guidelines comes from chapter 4 of Improving Web application security: Threats and Countermeasures:

- **Audit and log access across all of your Web application's tiers.** Use a combination of application level logging and platform auditing features, such as; IIS, Microsoft Operations Manager (MOM), SQL Server and Windows auditing.
- **Back up and analyze your Web application's log files regularly.** There's no point in logging activity if the log files are never analyzed. Log files should be removed from production servers on a regular basis. The frequency of removal is dependent upon your application's level of activity. Your design should consider the way that log files will be retrieved and moved to offline servers for analysis. Any additional protocols and ports opened on the Web server for this purpose must be securely locked down.
- **Consider identity flow.** Consider how your application will flow caller identity across multiple application tiers. You have two basic choices. You can flow the caller's identity at the operating system level using the Kerberos protocol delegation. This allows you to use operating system level auditing. The drawback with this approach is that it affects scalability because it means there can be no effective database connection pooling at the middle tier. Alternatively, you can flow the caller's identity at the application level and use trusted identities to access back end resources. With this approach, you have to trust the middle tier and there is a potential repudiation risk. You should generate audit trails in the middle tier that can be correlated with back end audit trails. For this, you must make sure that the server clocks are synchronized, although Microsoft Windows 2000 (and later) and Active Directory do this for you.
- **Log key events.** The types of events that should be logged include successful and failed logon attempts, modification and retrieval of data, network communications and administrative functions such as the enabling or disabling of logging. Logs should include the time and date of the event, the location of the event including the machine name, the identity of the current user, the identity of the process initiating the event and a detailed description of the event.
- **Secure your Web application's log files.** Secure log files using Windows ACLs and restrict access to the log files. This makes it more difficult for attackers to tamper with log files to cover their tracks. Restricting the access can be done by minimizing the number of individuals who can manipulate the log files. One recommended solution is to only authorize highly trusted accounts, such as administrators, access to the log files.

2.3.1.5 Auditing and logging countermeasures in ASP .NET 1.X and 2.0

A computer running Windows has, by default, the following three event logs:

- **Application log.** This log contains events, logged by application programs (applications) or programs. For example, a database program might record a file error in the application log. Application and program developers decide which events to monitor.

- **Security log.** The security log records events such as valid and invalid logon attempts, as well as events related to resource use such as creating, opening or deleting files or other objects. An administrator can specify what events are recorded in the security log. For example, if logon auditing is enabled, attempts to log on to the system are recorded in the security log.
- **System log.** The system log contains events logged by Windows system components. For example, the failure of a driver or other system component to load during startup is recorded in the system log. The event types logged by system components are predetermined by Windows.

Note though, that not all applications have full access rights to the default logs. Normal user programs have read and write access to the Application log, read only access to the System log and no access at all to the Security log. In addition, all users can create, read, write and delete custom event logs. Observe that it's possible for more privileged user programs to have greater access to the default logs.

The .NET class that let you interact with the Windows event logs is called `EventLog` and it's found in the `System.Diagnostics` namespace. With an `EventLog` object, you can read from existing logs, write entries to logs, create new custom logs, delete logs, create or delete event sources and respond to log entries.

To read from an event log you have to set the log name (`Log`) and machine name (`MachineName`) properties of the `EventLog` class object. The default values for these properties are an empty string (""), for the log name and "." (local computer) for the machine name. The following code snippet shows how to read from the Application event log on the local computer:

```
EventLog myLog = new EventLog("Application");
foreach(EventLogEntry entry in myLog.Entries)
{
    Console.WriteLine("\tEntry: " + entry.Message);
}
```

Note that you don't have to set the `MachineName` property, unless you are about to read from an event log on another computer. Once the `Log` and `MachineName` properties are specified, the `Entries` property is automatically populated with all the entries found in the specified event log, meaning that no explicit call to a read method is needed. When calling the `Entries` property, it returns an `EventLogEntryCollection` instance populated with `EventLogEntry` instances, one `EventLogEntry` for each entry found in the specified event log.

Observe two things; the first is that you have to include the `System.Diagnostics` namespace on top of the page to make the code snippet above work, the second is that custom log names is limited to eight characters by the system. Everything after that will be truncated so the names `Mylogname1` and `Mylogname2` are according to the system the same log.

Before you can write to an event log you have to register the application, or a component of your application, as a source for that event log. You do this by calling the `CreateEventSource` method of the `EventLog` class. This method is overloaded to accept the following one, two or three arguments:

- The first argument is the name of the source and it can be any random string, although it's recommended to use the name of your application or a component of your application.

Observe that the source must be unique, on the local computer, and only mapped to one event log at a time. If you decide to remap the source to a new log, you must reboot the computer for the change to take effect. However, a single event log can have many different sources writing to it simultaneously.

- The second argument is the name of the log. If the log that you specify doesn't exist on the computer, the system creates a custom log and registers your application as a source for that log.
- The third argument is the machine name of the computer the event log resides on. The default value is the local computer ("."), so the only time you need to use this third argument is when you intend to write to the event log of another computer.

An example of how to write to a custom event log, called `MyNewLog`, using the source `"MySource"` on the local computer is presented in the code snippet below:

```
//Create the source, if it does not already exist.
if(!EventLog.SourceExists("MySource"))
{
    EventLog.CreateEventSource("MySource", "MyNewLog");
    Console.WriteLine("CreatingEventSource");
}

//Create an EventLog instance and assign its source.
EventLog myLog = new EventLog();
myLog.Source = "MySource";

//Write an informational entry to the event log.
myLog.WriteEntry("Writing to event log.");
```

Now when you know how to read from and write to event logs, it's time to tackle the issue with event logs from an ASP .NET Web application. The default ASP .NET process identity for Web applications can write new records to the event log, but it doesn't have sufficient permissions to create new event sources. To address this issue you have to options:

- The first and recommended option is to create an application event source at installation time, when administrator privileges are available.

You do this by right click your project in the Solution Explorer window in VS .NET, point to add and then click Add Component. Select Installer Class from the list of templates and provide a suitable class file name. This creates a new installer class annotated with the `RunInstaller(true)` attribute as shown in the following code snippet:

```
RunInstaller(true)
public class EventSourceInstaller : System.Configuration.Install.Installer{ }
```

After that, you display the new installer class in Design view and then you display the Toolbox and click Components. Drag an `EventLogInstaller` component onto the Designer work surface and set the following `EventLogInstaller` properties:

- **Log.** Set this property to the name of the event log you would like to use. If you not intend to use a custom event log, you should set this property to "Application".
- **Source.** Set this property to your event source name. It's recommended to use either your application's name or a component of your application.

The next step is to build your project and then create an instance of this installer class at installation time.

Installer class instances are automatically created and invoked if you use a .NET Setup and Deployment project to create a Windows installer file (.msi). If you are not using .msi deployment, you should use the `InstallUtil.exe` system utility to create an instance of the installer class and execute it.

To confirm the successful generation of the event source, use the registry editor (`regedit.exe`) and navigate to:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\EventLog\Application\{source}
```

Once there, confirm that the key exists and that it contains an `EventMessageFile` string value that points to the default .NET Framework event message file:

```
\Windows\Microsoft.NET\Framework\{version}\EventLogMessages.dll
```

- The second option is to configure the permissions on the `EventLog` registry key to allow the ASP .NET process identity to create event sources at run time. You do this by granting the ASPNET process account, in Windows XP Professional, and the `NetWork Service` account, in Windows Server 2003, permission to the following registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\EventLog
```


The security events that aren't logged by default can easily be enabled, by reconfiguring the health monitoring feature's <healthMonitoring> element. But before presenting how to do this, let's go through all the built in security events possible to log in ASP .NET 2.0.

2.3.1.5.1 The built in security events that are logged by default

The following built in security events are, by default, logged to the Windows event log provider (EventLogWebEventProvider):

- The following major codes are logged for the **WebAuthenticationFailureAuditEvent** event type:
 - **AuditFormsAuthenticationFailure** (code 4005). This is used to identify brute force or dictionary attacks when using forms authentication classes. This major code may also be accompanied by one of the following detail codes:
 - **ExpiredTicketFailure**. This is used to identify cookie replay attacks.
 - **InvalidTicketFailure**. This is used to identify authentication cookie tampering.
 - **AuditMembershipAuthenticationFailure** (code 4006). This is used to identify brute force or dictionary attacks when using the membership feature.
- The following major event code is logged for exceptions of type **WebViewStateFailureAuditEvent**:
 - **AuditInvalidViewStateFailure** (code 4009). This is used to identify tampering with ViewState.
- The following major event codes are logged for events of type **WebFailureAuditEvent**, which are associated with authorization failures:
 - **AuditFileAuthorizationFailure** (code 4008). This is used to identify attempts to access unauthorized files or folders.
 - **AuditUnhandledAccessException** (code 4011). This is used to identify attempts of unauthorized access to resource.
 - **AuditUnhandledSecurityException** (code 4010). This is used to identify attempts to perform actions not allowed by the current trust level.
 - **AuditUrlAuthorizationFailure** (code 4007). This is used to identify attempts to access an unauthorized path or page.
- The following major event codes are logged for exceptions of type **WebErrorEvent**, which are associated with compilation or configuration errors and may indicate unauthorized alteration of the content of your Web site:
 - **WebErrorCompilationError** (code 3007). This indicates that an error occurred during application compilation.
 - **WebErrorConfigurationError** (code 3008). This indicates that a configuration error occurred.
 - **WebErrorObjectStateFormatterDeserializationError** (code 3011). This indicates that an error deserializing internal state objects occurred.
 - **WebErrorOtherError** (code 3009). This indicates that an unclassified error occurred.
 - **WebErrorParserError** (code 3006). This indicates that a parser error occurred.
 - **WebErrorPropertyDeserializationError** (code 3010). This indicates that an error deserializing internal state objects occurred.

- The following major event codes are logged for exceptions of type **WebRequestErrorEvent**, which are associated with runtime errors, and may indicate an attack on your Web site:
 - **DiskOutputCacheInformation** (code 5003). This indicates that a disk output cache event occurred. This event is always accompanied by a detail code giving more information.
 - **DiskOutputCacheQuotaExceeded** (code 5001). This indicates that the disk output cache quota was exceeded.
 - **RuntimeErrorPostTooLarge** (code 3004). This indicates that the size of the posted information exceeded the allowed limits.
 - **RuntimeErrorRequestAbort** (code 3001). This indicates that the Web request has been aborted.
 - **RuntimeErrorUnhandledException** (code 3005). This indicates that an unhandled exception occurred.
 - **RuntimeErrorValidationFailure** (code 3003). This indicates that a validation error occurred.
 - **RuntimeErrorViewStateFailure** (code 3002). This indicates that a view state failure occurred.

2.3.1.5.2 The built in security events that are not logged by default

Although the following default events aren't logged by default, they are recommended to enable to aid investigation in the case of an attack:

- The following major codes can be logged for the **WebApplicationLifetimeEvent** event type for detecting application availability:
 - **ApplicationStart** (code 1001) / **ApplicationShutdown** (code 1002). These events indicate application startup and shutdown and they are always accompanied by a detail code giving more information. If you see a large quantity of these events, your application may be the target of a denial of service attack.
 - **ApplicationCompilationStart** (code 1003) / **ApplicationCompilationEnd** (code 1004). These events indicate the compilation of the application has started and finished. If you see these events, it may be an indication of unauthorized modification of your application content.
- The following major codes can be logged for the **WebAuthenticationSuccessAuditEvent** event type:
 - **AuditFormsAuthenticationSuccess** (code 4001). This is used to maintain an audit trail of a successful forms authentication, which can then be retraced if the system is compromised, when using the forms authentication classes.
 - **AuditMembershipAuthenticationSuccess** (code 4002). This is used to maintain an audit trail of a successful membership authentication, which can then be retraced if the system is compromised, when using the new ASP .NET 2.0 membership feature.
- The following major codes can be logged for the **WebSuccessAuditEvent** event type:
 - **AuditFileAuthorizationSuccess** (code 4004). This is used to maintain an audit trail, which can then be retraced to identify all successful file accesses by an attacker if the system is compromised.
 - **AuditUrlAuthorizationSuccess** (code 4003). This is used to maintain an audit trail, which can then be retraced to identify all successful URL and path accesses by an attacker if the system is compromised.

- The following major codes can be logged for the **WebHeartbeatEvent** event type, which provide an audit trail that monitors normal operation of your application:
 - **ApplicationHeartbeat** (code 1005). This is used to log a heartbeat for the application. The heartbeat interval is defined by the `heartBeatInterval` attribute of the `<healthMonitoring>` element in the root `Web.config` file. The default value is 0, indicating no heartbeat.
- The following major codes can be logged for the **WebRequestEvent** event type, which provide an audit trail of normal request completion and abort:
 - **RequestTransactionComplete** (code 2001). This indicates the Web request was completed.
 - **RequestTransactionAbort** (code 2002). This indicates the Web request was aborted.

2.3.1.5.3 Custom security events

There are additional security events that you can instrument to improve your ability to detect and understand attacks on your application. The following security events are recommended to log in addition to the default security events:

- **Authorization events.** The following list of additional authorization events represents a list of scenarios for which it's recommended that you create custom security events in order to improve visibility on the health of your application:
 - **Accessing objects or keys in memory.** This is used to instrument success and failure events that can be used to identify objects or keys compromise.
 - **Roles authorization failure events.** This is used to identify attempts at unauthorized access.
 - **Roles authorization success events.** This is used to maintain audit trails of user activities.
- **Session management events.** By instrumenting the following events you can aid your ability to track user activities and detect anomalies in user behavior:
 - Session creation
 - Session expiration
 - Session lifetime events
 - Session termination
 - Session timeout
- **User management events.** Instrumenting the following events can aid your ability to track anomalies in user account changes:
 - Account creation / deletion / lockout / modification.
 - Password resets / changes.
 - Roles assignment.

2.3.1.5.4 How to use the health monitoring feature, introduced in ASP .NET 2.0

The best way to understand how to use health monitoring, in your own Web application, is to create your own custom Web event. But before we do that, we'll look at the default event providers.

2.3.1.5.4.1 Web event providers

Health monitoring, in ASP .NET version 2.0, supports an event provider model. Event providers encapsulate the underlying event stores and provide a consistent API. The following built in Web event providers are, by default, available in ASP .NET 2.0:

- **SimpleMailWebEventProvider.** This provider sends email for event notifications.
- **TemplatedMailWebEventProvider.** This provider uses templates to define and format email messages sent for event notifications.
- **SqlWebEventProvider.** This provider logs event details to a SQL Server database. If you use this provider, you should encrypt the connection string in your Web.config file by using the Aspnet_regiis.exe tool.
- **EventLogWebEventProvider.** This provider logs events to the Windows application event log. This provider is the default provider in ASP .NET 2.0.
- **TraceWebEventProvider.** This provider logs events as ASP .NET trace messages.
- **WmiWebEventProvider.** This provider maps ASP .NET health monitoring events to Windows Management Instrumentation (WMI) events.

If none of the above Web event providers are adequate, it's possible to create custom Web event providers. You create a custom Web event provider by creating a class that either derives from one of the Web event providers above or from one of Web event provider base classes found in the System.Web.Management namespace; BufferedWebEventProvider or WebEventProvider.

2.3.1.5.4.2 How to create your own custom security event

Now it's time to create a custom Web event. This will be done in the four steps:

1. The first step is to create a new class library, which you name MyWebEvents. Then you rename the Class1.cs class file to MyCriticalEvent.cs. After that you add the following code to the MyCriticalEvent.cs file:

```
using System.Web;
using System.Web.Management;

namespace MyWebEvent
{
    public class MyCriticalEvent:WebAuditEvent
    {
        public MyCriticalEvent(string msg, object eventSource, int
eventCode):base(msg, eventSource, eventCode)
        {}

        public MyCriticalEvent(string msg, object eventSource, int
eventCode, int eventDetailCode):base(msg, eventSource, eventCode,
eventDetailCode)
        {}

        public override void FormatCustomEventDetails(WebEventFormatter
formatter)
        {
            base.FormatCustomEventDetails(formatter);
            formatter.AppendLine("Here you place your event text");
            formatter.AppendLine("Even more text");
        }
    }
}
```

First are the MyCriticalEvent class derived from the WebAuditEvent class. Secondly are two appropriate public constructors created and finally are the FormatCustomEventDetails class method overridden to supplement the standard event output with custom data.

Now it's time to compile the assembly. If you want to use the assembly in multiple applications, you should sign the assembly with a strong name and install it in the global assembly cache, otherwise build the class library using the following command at the command prompt:

```
csc /t:library thenameofyourclasslibrary.cs
```

The final step is then to add the resulting Dynamic Link Library (DLL) file to your application's Bin folder. If you don't have a Bin folder, right click your project and hover over add folder and then click add Bin folder.

2. The second step is to create a simple ASP .NET Web application, which we'll monitor and instrument your custom event.

Use VS .NET 2005 to create a new Web application.

After that, add references to the newly created Web event and to the WebExtendedBase integer constant. The reason why you want to include the WebExtendedBase constant is to make sure that you specify an event code greater than the WebExtendedBase value when you raise your custom Web event. Every value up to WebExtendedBase is reserved for system generated events. You do this by including the following using directives on top of your Default.aspx.cs file:

```
using System.Web.Management;  
using MyWebEvent;
```

When you're done with that it's time to instrument your application to use your newly created Web event. The ASP .NET runtime is only instrumented to raise standard events at the appropriate time, meaning that you must instrument your Web events by yourself.

The easiest way to do this is to add a button to your application's Default.aspx page and add the following code to the button click event handler:

```
protected void Button1_Click(object sender, EventArgs e)  
{  
    MyCriticalEvent testEvent = new MyCriticalEvent("Critical Operation  
        Performed", this, WebEventCodes.WebExtendedBase + 1);  
  
    testEvent.Raise();  
}
```

The code creates a new custom event object of type MyCriticalEvent and then calls the testEvent's Raise method to fire the event.

3. The third step is to configure your application for health monitoring. You do this by configuring the <healthMonitoring> element in the root Web.config file. The easiest way to understand how to do this is to go through the <healthMonitoring> element's sub elements. So here they are:

- **<bufferModes>**. This element configures the buffering properties of the providers that inherit from System.Web.Management.BufferedWebEventProvider, which includes the MailWebEventProvider "base class" and SqlWebEventProvider. You can also derive custom Web event providers from the base class and use buffer modes.

You configure buffering to minimize the performance impact and overhead of recording events. You can use the <bufferModes> configuration to define how long events are buffered before they are written to the Web event provider and you can distinguish between urgent, critical and regular events.

You can use any of the default buffer modes; Critical Notification, Notification, Analysis and Logging, which are configured in the machine level Web.config.default file, or configure a custom buffer mode.

To use buffering you need to set the buffer attribute, on your Web event provider configuration, to "true" and to use a specific buffering configuration use the bufferMode attribute on your provider configuration. The following example is a simplified version of how it may look like:

```

<healthMonitoring>
  <providers>
    <add name="Name" buffer="true" bufferMode="Logging"/>
  </providers>
</healthMonitoring>

```

To configure a custom buffer mode, add the following to your applications Web.config file:

```

<healthMonitoring>
  <bufferModes>
    <add
      name="Extra Critical Notification"
      <!--
      This is a name for the buffer mode used to reference it from
      other elements, such as the bufferMode attribute in the
      <provider> element.
      -->
      maxBufferSize="10"
      <!--
      This is the maximum number of events that can be buffered,
      by a provider, before flushing and writing them to a data store.
      -->
      maxFlushSize="5"
      <!--
      This is the maximum number of events per flush. Its value
      should be between 1 and maxBufferSize.
      -->
      urgentFlushThreshold="1"
      <!--
      This is the minimum number of events after which the events
      should be flushed. Its value should be less then or equal to
      maxBufferSize.
      -->
      regularFlushInterval="Infinite"
      <!--
      This is the time interval per flush. Its value cannot be zero.
      -->
      urgentFlushInterval="00:01:00"
      <!--
      This is the minimum time between flushes. Its value must be
      between 0 and regularFlushInterval.
      -->
      maxBufferThreads="1"
      <!--
      This is the maximum number of threads used for flushing.
      -->
    />
  </bufferModes>
</healthMonitoring>

```

- **<providers>**. You use the <providers> element to indicate what logging sinks are available for logged events. Note that any event providers you configure are not actually used for reporting events until you configure an event rule that specifies a configured provider. For more information, see <rules> section below.

The default configuration in the root Web.config file defines the following providers:

- **EventLogWebEventProvider**. This provider uses the EventLogWebEventProvider class to log to the Windows application event log.
- **SqlWebEventProvider**. This provider uses the SqlWebEventProvider class for logging to a SQL Server or SQL Server Express instance.
- **WmiWebEventProvider**. This provider uses the WmiWebEventProvider class for logging to WMI.

You can use any of these default event providers in your own health monitoring configuration or you can configure new providers using any of the standard Web event provider classes. You can also use any custom Web event provider class that derives from the `WebEventProvider` base class.

If you decide to use the `SqlWebEventProvider` event provider you must create the database that will be used, configure the connection string element and configure the `<providers>` element.

Creating the database is really easy since this is almost already done for you. The only thing you need to is to run the `aspnet_regsql.exe` tool, which can be found in the `%SystemRoot%\Microsoft.NET\Framework\v2.0.50215` folder. The tool then creates and configures SQL server instance with everything necessary for the `SqlWebEventProvider` to work properly.

If you also have decided to use a SQL Server Express instance you don't have to configure the connection string since the default connection string, in the machine level `machine.config` file, is set to use the SQL Server Express as you can see below:

```
<connectionStrings>
  <add name="LocalSqlServer" connectionString="data
source=.\SQLEXPRESS;Integrated Security=SSPI;
AttachDBFilename=|DataDirectory|aspnetdb.mdf; User
Instance=true" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

If you decided to use a SQL Server instance you must change the connectionstring to the following:

```
<connectionStrings>
  <add name="LocalSqlServer" connectionString="data
source=(local); Integrated Security=SSPI;
Database=DatabaseName"/>
</connectionStrings>
```

The last step is to configure the `<healthMonitoring>` element's `<providers>` sub element to something like this in the `web.config` file:

```
<providers>
  <add
    connectionStringName="LocalSqlServer"
    <!--
    If you are using the SqlWebEventProvider, use this attribute to
    specify the friendly name of the connection string used for connecting
    to the SQL Server database.
    -->
    maxEventDetailsLength="1073741823"
    <!--
    This is the maximum length of the event details.
    -->
    buffer="true"
    <!--
    If you are using the SqlWebEventProvider, use this attribute to enable
    event buffering. If this attribute is true, you must configure the
    bufferMode attribute. The default value is false.
    -->
    bufferMode="Extra Critical Notification"
    <!--
    If you are using the SqlWebEventProvider, use this attribute to
    specify the name of the buffer mode to be used for buffering the
    events.
    -->
    name="SqlWebEventProvider"
    <!--
    This attribute specifies the provider name.
    -->
```

```

type="System.Web.Management.SqlWebEventProvider, System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"

```

```

<!--

```

```

This is a fully qualified assembly reference to the provider class. This class should implement the System.Configuration.Provider.ProviderBase class.
-->

```

```

-->

```

```

/>

```

```

</providers>

```

- **<profiles>**. You use the <profiles> element to specify sets of parameters to use when configuring events. These parameters indicate the minimum number instances after which the event should be logged, the maximum number of instances and the minimum interval between logging two similar events. This element can be critical in controlling the amount of information generated by defining when monitoring begins and when it ends by setting thresholds.

You can use this element to throttle the event occurrences. It can help prevent an attack against the eventing system itself or an event sink such as SQL Server or the Windows event log. The following code example shows the default settings from the root Web.config.default file

```

<profiles>

```

```

  <add name="Default" minInstances="1" maxLimit="Infinite"
    minInterval="00:01:00"/>

```

```

  <add name="Critical" minInstances="1" maxLimit="Infinite"
    minInterval="00:00:00"/>

```

```

</profiles>

```

You can either use the default profiles in your own health monitoring configuration or you can create your custom new profiles. To create a custom profile, add the following <profiles> sub element within the existing <healthMonitoring> element of your web.config file:

```

<profiles>

```

```

  <add

```

```

    name="Throttle"

```

```

    <!--

```

```

    This is the name used to reference it from other elements.

```

```

    -->

```

```

    minInstances="1"

```

```

    <!--

```

```

    This is the minimum number of occurrences before an event is fired and logged.

```

```

    -->

```

```

    maxLimit="1000"

```

```

    <!--

```

```

    If you want to set a maximum limit after which the specific events should stop firing and logging, use this attribute. The default setting is Infinite.

```

```

    -->

```

```

    minInterval="00:00:01"

```

```

    <!--

```

```

    If you want to set a minimum time interval between logging same event again, use this attribute. The format is "hh:mm:ss" and the default is 00:00:00.

```

```

    -->

```

```

/>

```

```

</profiles>

```

- **<rules>**. Use the <rules> element to specify which events to log through which event provider. As an option, you can apply a profile to an event logging rule by specifying the name of a <profiles> entry or you can apply the same attributes used in the <profiles> definition directly to a <rule> definition.

The default rule settings are defined in the root `Web.config.default` file and it cause the "All Errors" and "Failure Audits" event mappings to be logged to the Windows event log.

```
<rules>
  <add name="All Errors Default" eventName="All Errors"
        provider="EventLogProvider" profile="Default"
        minInstances="1" maxLimit="Infinite"
        minInterval="00:01:00" custom=""/>
  <add name="Failure Audits Default" eventName="Failure Audits"
        provider="EventLogProvider" profile="Default"
        minInstances="1" maxLimit="Infinite"
        minInterval="00:01:00" custom=""/>
</rules>
```

You can disable the default rules mappings by using the `<clear>` child element inside the `<rules>` element, before adding your own rules.

To create a new event rule for the custom event you created earlier, add the following `<rules>` element inside the `<healthMonitoring>` section in your application's `Web.config` file:

```
<rules>
  <add
    name="Critical event"
    <!--
    This is the name used to reference it from other elements.
    -->
    eventName="My Critical Event"
    <!--
    This is the name of the event you want to monitor for which the rule
    is being configured.
    -->
    provider="SqlWebEventProvider"
    <!--
    Use this attribute to specify the name of the provider to use to log
    this event.
    -->
    profile="Throttle"
    <!--
    If you want to use a preconfigured profile, specify the name of the
    profile using this attribute. Note that if you specify a profile, the
    profile supplies the values for the minInstances, maxLimit and
    minInterval rules attributes.
    -->
    minInstances="1"
    <!--
    If you want to specify the minimum number of occurrences before an
    event is fired and logged, use this attribute. If you specify a profile
    attribute, this attribute overrides the minInstances value in the profile.
    -->
    maxLimit="Infinite"
    <!--
    If you want to set a maximum limit after which the specific events
    should stop firing and logging, use this attribute. The default setting
    is Infinite. If you specify a profile attribute, this attribute overrides
    the maxLimit value in the profile.
    -->
    minInterval="00:01:00"
    <!--
    If you want to set a minimum time interval between logging the same
    event again, use this attribute. The format is "hh:mm:ss" and the
    default is 00:00:00. If you specify an equivalent profile attribute, this
    attribute overrides the minInterval value in the profile.
    -->
  />
</rules>
```

- **<eventMappings>**. Event mappings are named groups of events that you want to monitor. Note that you can include a particular event in more than one named group. The default event mappings include two high level groupings; All Events and All Audits, which include all events and all audits respectively. There are also subsets of each of those groupings. For example, All Errors includes all error events, and Failure Audits includes all audit failure events. You can review the default settings in the machine level Web.config.default file.

You can use the default event mappings in your own health monitoring configuration or you can create new event mappings. You must create event mappings for any custom Web events you create. A custom event mapping may look something like this:

```
<eventMappings>
  <add
    name="My Critical Event"
    <!--
    This is the name used to reference it from other elements.
    -->

    type="MyWebEvents.MyCriticalEvent, MyWebEvent"
    <!--
    This is a fully qualified assembly reference to the event class.
    -->
    startEventCode="0"
    <!--
    If you want to map events of a similar type in a specific range of
    event codes, use this attribute to set the starting event code for the
    range of events codes. The default setting is 0.
    -->

    endEventCode = "Infinite"
    <!--
    If you want to map events of a similar type in a specific range of
    event codes, use this attribute to set the top end of the range of
    event codes. The default setting is Infinite.
    -->
  />
</eventMappings>
```

4. The final step is then to run and test your Web application. You do this by building your Web application and click the button on the Default.aspx page. After that you open the aspnet_WebEvent_Events table in the SQL server database you just created in step 3, using the aspnet_regsql.exe tool. The details column will then contain the information from the event you just triggered.

2.3.2 Authentication and authorization

Authentication is the process of determining a user's identity. There are three aspects to consider:

- Identify where authentication is required in your Web application. It's generally required whenever a trust boundary, such as an assembly, process or host, is crossed.
- Validate who the caller is. Users authenticate themselves with usernames and passwords.
- Identify the user on subsequent requests. This requires some form of authentication token.

If the authentication mechanism isn't correctly chosen and implemented, it can expose vulnerabilities that attackers can exploit to gain access to your system.

Authorization on the other hand is the process of determining what an authenticated user can or can't do. Improper or weak authorization can lead to information disclosure or data tampering.

The top threats that either exploits an authentication or authorization vulnerability are:

- Authentication: Brute force attacks

- Authentication: Cookie replay
- Authentication: Credential theft
- Authentication: Dictionary attacks
- Authentication: Network eavesdropping
- Authorization: Data tampering
- Authorization: Disclosure of confidential data
- Authorization: Elevation of privilege
- Authorization: Luring attacks

2.3.2.1 Authentication threat: Brute force attacks

Brute force attacks rely on computational power to crack hashed passwords or other secrets secured with hashing and encryption. To mitigate the risk, use strong passwords.

2.3.2.2 Authentication threat: Cookie replay attacks

With this type of attack, the attacker captures the user's authentication cookie using monitoring software and replays it to the application to gain access under a false identity.

Countermeasures to prevent cookie replay include:

- Use an encrypted communication channel provided by SSL whenever an authentication cookie is transmitted.
- Use a cookie timeout to a value that forces authentication after a relatively short time interval. Although this doesn't prevent replay attacks, it reduces the time interval in which the attacker can replay a request without being forced to reauthenticate because the session has timed out.

2.3.2.3 Authentication threat: Credential theft

If your application implements its own user store containing user account names and passwords, compare its security to the credential stores provided by the platform, for example a Microsoft Active Directory directory service or Security Accounts Manager (SAM) user store. Browser history and cache also store user login information for future use. If the terminal is accessed by someone other than the user who logged on and the same page is hit, the saved login will be available.

Countermeasures to help prevent credential theft include:

- Use and enforce strong passwords.
- Store password verifiers in the form of one way hashes with added salt.
- Enforce account lockout for end user accounts after a set number of retry attempts.
- To counter the possibility of the browser cache allowing login access, create functionality that either allows the user to choose to not save credentials or force this functionality as a default policy.

2.3.2.4 Authentication threat: Dictionary attacks

This attack is used to obtain passwords. Most password systems don't store plaintext passwords or encrypted passwords. They avoid encrypted passwords because a compromised key leads to the compromise of all passwords in the data store. Observe that the loss of a key, means that all passwords in the data store is invalidated.

Most user store implementations hold password hashes. Users are authenticated by recomputing the hash based on the user supplied password value and comparing it against the hash value stored in the database. If an attacker manages to obtain the list of hashed passwords, a brute force attack can be used to crack the password hashes.

With the dictionary attack, an attacker uses a program to iterate through all of the words in one or more dictionaries and computes the hash for each word. The resultant hash is compared with the value in the data store. Weak passwords such as "SonyEricsson" will therefore be cracked quickly. Stronger passwords such as "?You'LiNevaFiNdMe!", are less likely to be cracked.

Observe that once the attacker has obtained the list of password hashes, the dictionary attack can be performed offline and does not require interaction with the application.

Countermeasures to prevent dictionary attacks include:

- Use strong passwords that are complex, are not regular words and contain a mixture of upper case, lower case, numeric and special characters.
- Store non reversible password hashes in the user store. It's also recommended to combine a salt value, a cryptographically strong random number, with the password hash.

2.3.2.5 Authentication threat: Network eavesdropping

If authentication credentials are passed in plaintext from client to server, an attacker armed with rudimentary network monitoring software on a host on the same network can capture traffic and obtain user names and passwords.

Countermeasures to prevent network eavesdropping include:

- Use authentication mechanisms that do not transmit the password over the network such as Kerberos protocol, Forms or Windows authentication.
- If you must transmit passwords over the network, make sure passwords are encrypted or use an encrypted communication channel such as Secure Socket Layer (SSL).

2.3.2.6 Authorization threat: Data tampering

Data tampering refers to the unauthorized modification of data.

Countermeasures to prevent data tampering include:

- Use strong access controls to protect data in persistent stores to ensure that only authorized users can access and modify the data.
- Use role based security to differentiate between users who can view data and users who can modify data.

2.3.2.7 Authorization threat: Disclosure of confidential data

The disclosure of confidential data can occur if sensitive data can be viewed by unauthorized users. Confidential data includes application specific data such as credit card numbers, employee details, financial records and so on together with application configuration data such as service account credentials and database connection strings. To prevent the disclosure of confidential data you should secure it in persistent stores such as databases and configuration files and during transit over the network. Only authenticated and authorized users should be able to access the data that is specific to them. Access to system level configuration data should be restricted to administrators.

Countermeasures to prevent disclosure of confidential data include:

- Perform role checks before allowing access to the operations that could potentially reveal sensitive data.
- Use strong ACLs to secure Windows resources.

- Use standard encryption to store sensitive data in configuration files and databases.

2.3.2.8 Authorization threat: Elevation of privilege

When you design an authorization model, you must consider the threat of an attacker trying to elevate privileges to a powerful account such as a member of the local administrators group or the local system account. By doing this, the attacker is able to take complete control over the application and local machine. For example, with classic ASP programming, calling the RevertToSelf API from a component might cause the executing thread to run as the local system account with the most power and privileges on the local machine.

The main countermeasure that you can use to prevent elevation of privilege is to use least privileged process, service and user accounts.

2.3.2.9 Authorization threat: Luring attacks

A luring attack occurs when an entity with few privileges is able to have an entity with more privileges perform an action on its behalf.

To counter this threat, you must restrict access to trusted code with the appropriate authorization methods. Using .NET Framework code access security helps in this respect by authorizing calling code whenever a secure resource is accessed or a privileged operation is performed.

2.3.2.10 Design guidelines for preventing authentication attacks

Many Web applications use a password mechanism to authenticate users, where the user supplies a username and password in an HTML form. The issues and questions to consider when implementing authentication are the following (Improving Web application security: Threats and Countermeasures):

- **Are usernames and passwords sent in plaintext over an insecure channel?** If so, can an attacker easily eavesdrop, with a network monitoring software tool, and capture the credentials. The countermeasure is to secure the communication channel by using SSL.
- **How are the credentials stored?** If you are storing user names and passwords in plaintext you're inviting trouble. What if your application directory is improperly configured and an attacker browses to the file and downloads its contents or adds a new privileged logon account? What if a disgruntled administrator copies your "authentication database"?
- **How are the credentials verified?** There is no need to store user passwords if the sole purpose is to verify that the user knows the password value. Instead, you can store a verifier in the form of a hash value and recompute the hash using the user supplied value during the logon process. To mitigate the threat of dictionary attacks against the credential store, use strong passwords and combine the password hash with a randomly generated salt value.
- **How is the authenticated user identified after the initial logon?** The answer is that some form of authentication ticket is required. This authentication ticket is normally stored in a cookie. If it's sent across an insecure channel, an attacker can capture the cookie and use it to access the application. A stolen "authentication cookie" is a stolen logon.

The following practices improve your Web application's authentication:

- **Be able to disable accounts.** If the system is compromised, being able to deliberately invalidate credentials or disable accounts can prevent further and / or additional attacks.
- **Don't send passwords over the wire in plaintext.** Plaintext passwords sent over a network are vulnerable to eavesdropping. To address this threat, secure the communication channel by using SSL to encrypt the traffic.
- **Don't store passwords in user stores.** If you must verify passwords, you don't need to actually store them. Instead, store a one way hash value and then recompute the hash using the user supplied password. To mitigate the threat of dictionary attacks against the user store, use strong passwords and incorporate a random salt value with the password.

- **Protect authentication cookies.** A stolen authentication cookie is a stolen logon. Protect authentication tickets using encryption and secure communication channels. Also limit the time interval in which an authentication ticket remains valid. Although reducing the cookie timeout doesn't prevent replay attacks, it does limit the amount of time the attacker has to access the site using the stolen cookie.
- **Require strong passwords.** Don't make it easy for attackers to crack passwords. A general practice is to require a minimum of eight characters and a mixture of uppercase and lowercase characters, numbers and special characters. Observe that a strong password is crucial to counter brute force attacks.
- **Separate public and restricted areas.** It's recommended that you separate your Web site into public and restricted areas. The public areas should be accessible to everyone, while the restricted areas should only be accessible to authenticated users.

The benefit of partitioning your Web site into public and restricted areas is that you can apply separate authentication and authorization rules across the Web site and thereby limit the use of SSL to the areas that require authenticated access.

- **Support password expiration periods.** It's recommended that every user should be forced to change their password after a preset expiration period.
- **Use account lockout policies for end user accounts.** Disable end user accounts or write events to a log after a preset number of failed login attempts. If you are using Windows authentication, such as NTLM or the Kerberos protocol, these policies can be configured and applied automatically by the operating system. With Forms authentication, these policies are the responsibility of the Web application developer.

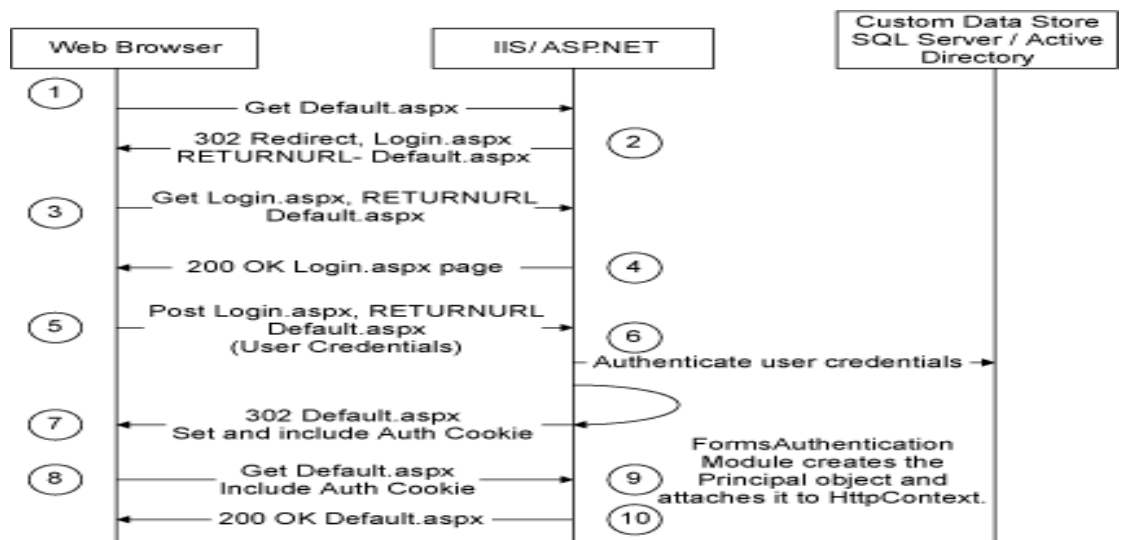
Be careful that account lockout policies cannot be abused in denial of service attacks. For example, well known default service accounts such as IUSR_MACHINENAME should be replaced by custom account names to prevent an attacker who obtains the IIS Web server name from locking out this critical account.

2.3.2.11 Authentication and authorization countermeasures in ASP .NET 1.X

There are several ways to authenticate users in ASP .NET Web applications. The most common way in Web applications is to use Forms authentication. Therefore will the focus in this heading be on Forms authentication.

2.3.2.11.1 Forms authentication

The sequence of events triggered by an unauthenticated user, when using forms authentication, who tries to access a secured file or resource are presented in the figure below, which has been taken from chapter 8 of Building Secure ASP.NET applications:



Let's look a little closer on the events shown in the figure above:

1. The first event is triggered when an anonymous user issues a request for the page default.aspx.

To understand what happens next it's important that you know that forms authentication is a two step process. The first step is when IIS authenticates the user and creates a Windows token to represent the user. IIS determines the authentication mode that it should use for a particular application by looking at its metabase settings. If IIS is configured to use anonymous authentication is a Windows token, for the IUSR_MACHINE account (the default account), generated and used to represent the anonymous user. IIS then passes the token to the ASP .NET runtime, which performs the second step of the forms authentication process.

Observe that forms authentication doesn't rely on IIS authentication, you should configure anonymous access for your application in IIS if you intend to use forms authentication in your ASP .NET application

The ASP .NET runtime then checks the <authorization> element, of the Web application's Web.config file, and finds a <deny users="?> element, meaning that only authenticated users are allowed.

2. The first thing the ASP .NET runtime do is to look for an authentication cookie. If it fails, is the user redirected to the URL of the login page (Login.aspx in our case), specified by the LoginUrl attribute of the <forms> element in the Web.config file.

The user will, in a later event, supply its username and password and submit through this form. Information about the originating page is placed in the query string using RETURNURL string as the key. The server HTTP reply is as follows:

302 Found Location:

http://localhost/RestrictedFolder/login.aspx?RETURNURL=%2fdefault.aspx

3. The browser requests the Login.aspx page and includes the RETURNURL parameter in the query string.
4. The server returns the logon page and the 200 OK HTTP status code.
5. The user enters hers or his credentials on the logon page and posts the page, including the RETURNURL parameter from the query string, back to the server.
6. The credentials are validated against a store like SQL Server or Active Directory and the user's roles are retrieved.
7. A cookie is created and sent back to the client. The cookie contains the user's roles. By storing the roles in the ticket, you avoid having to retrieve the roles from the database for every Web request from the same user.

For the authenticated user, the server redirects the browser to the original URL that was specified in the query string by the RETURNURL parameter. The server HTTP reply is as follows:

302 Found Location:

http://localhost/TestSample/default.aspx

8. The browser requests the default.aspx page again. This request includes the forms authentication cookie.
9. At the server is event 1 triggered again. But this time the user is authenticated. Instead of redirecting the user to the login page is the request redirected to the Application_AuthenticateRequest event handler, which you find in the file global.asax. This event handler uses the authentication ticket found in the authentication cookie to create an IPrincipal object, which are stored in the HttpContext.User object.
10. After that the server has verified the authentication cookie, it grants access and returns the default.aspx page

2.3.2.11.1.1 Recommended development steps for Forms authentication

The list below highlights the key steps that you must perform to implement Forms authentication:

1. Configure IIS for anonymous access.
2. Configure ASP .NET for both Forms authentication and authorization.
3. Create a login Web form, authenticate the user, retrieve the user's roles and create an authentication ticket in the login buttons click event handler.
4. Create an IPPrincipal object and put the IPPrincipal object into the current HTTP context and use it to authorize the user based on its role(s).

2.3.2.11.1.1.1 Configure IIS for anonymous access

Configuring your application's virtual directory, in IIS, for anonymous access, is a 3 step process:

1. Start the Internet Information Services administration tool.
2. Right click your application's virtual directory and chose Properties. After that click Directory Security.
3. In the Anonymous access and authentication control group, click Edit and select Anonymous access.

2.3.2.11.1.1.2 Configure ASP .NET for both Forms authentication and authorization

To easiest way to start configuring ASP .NET for both Forms authentication and authorization is to use the Solution Explorer, within VS .NET, to open the file Web.config. Once Web.config is open, you locate the <authentication> element and change it to the following:

```
<authentication mode="Forms">
  <forms      name="NameOnAuthCookie"
    <!--
    The name attribute specifies the name of the HTTP cookie used for
    authentication. The default name is .ASPXAUTH, but it's recommended to
    use an unique name. By using unique names you ensure that users only are
    authenticated against one specific Web application if you have multiple Web
    applications on your server.
    -->
    loginUrl="RestricedFolder\LoginPage.aspx"
    <!--
    The loginUrl attribute specifies the login page's URL, where all
    unauthenticated users are redirected.
    -->
    requireSSL="true"
    <!--
    The requireSSL attribute indicates whether a secure connection (SSL) is
    required to transmit the authentication cookie. The default value is false. If
    true, ASP .NET sets the secure attribute for the authentication cookie and a
    compliant browser doesn't return the cookie unless the connection is using
    SSL. Therefore it's important to make sure that your Web server supports
    HTTPS in the folder in which the login page resides. This attribute isn't
    available in ASP .NET 1.0.
    -->
    protection="All"
    <!--
    The protection attribute indicates how the application intends to protect the
    authentication cookie. The available values are; All, Encryption, None and
    validation. The default and recommended value are All, meaning that both
    encryption and validation will be used. The algorithms that are possible to
    use are presented in the Cryptography section.
    -->

    timeout="20"
```

```

<!--
The timeout attribute specifies the cookie expiration time in minutes. The
default value is 30. A recommended value is 20.
-->
path="/ApplicationName"
<!--
The path attribute specifies the path for the authentication cookie issued by
the application. The default value is slash (/), but it's recommended that you
use an unique application path to ensure that users only are authenticated
against one specific Web application if you have multiple Web applications on
your server.
-->
slidingExpiration="true"
<!--
The slidingExpiration attribute indicates whether the sliding expiration is
enabled. This attribute isn't available in ASP .NET 1.0.
-->
>
</forms>
</authentication>

```

Once you're done with the <authentication> element it's time to configure the <authorization> element. But before you do that, make sure that you partition your Web site. You do this by placing all your secure pages in a subdirectory that only authenticated users have access to.

To ensure that SSL is used when accessing a file in the "Secure" folder, you have to configure the secure folder in IIS to require SSL. You configure a folder to require SSL by starting Internet Information Services and follow these steps:

1. Expand your server name and Web Site. Observe that this must be a Web site that has a certificate installed. On how to install a certificate see page 479 – 483 in Building Secure ASP.NET applications.
2. Right click your "Secure" folder and then choose Properties.
3. On the Directory Security tab, under secure communications, click edit.
4. In the Secure Communications dialog box, select the "Require secure channel (SSL)" check box.

This sets the AccessSSL = true attribute for the folder in the IIS metabase. Requests for pages in the secured folder will only be successful if https is used on the request URL.

Notice that navigating between the public (http) and restricted (https) pages of your Web site is an issue because a redirect always uses the protocol of the current page and not the target page.

Once a user logs on and browses pages in a directory that is secured with SSL, relative links such as "..\publicpage.aspx" or redirects to HTTP pages result in the pages being served using the https protocol, which incurs an unnecessary performance overhead. To avoid this, use absolute links such as "http://servername/appname/publicpage.aspx" when redirecting from an HTTPS page to an HTTP page.

Similarly, when you redirect to a secure page from a public area of your site, you must use an absolute HTTPS path, such as "https://servername/appname/secure/login.aspx", rather than a relative path, such as restricted/login.aspx.

Now it's time to configure the <authorization> element, in your Web application's Web.config file, to allow anonymous access to all of your Web applications public pages. You do this by using the following <authorization> element:

```

<system.web>
  <authorization>
    <allow users = "*" />
  </authorization>
</system.web>

```

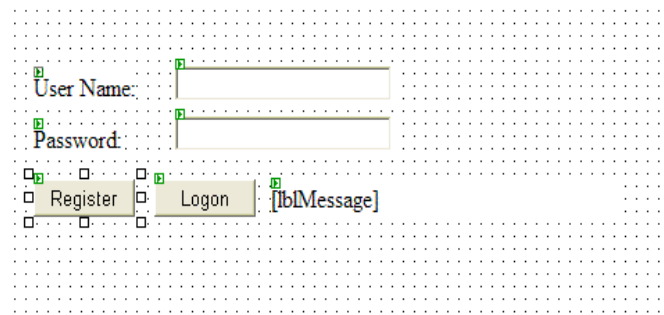
Both this and the example below are examples of URL authorization.

After you're done with that you use the following <authorization> element, inside a <location> element, to deny access to unauthenticated users and force a redirect to the login page that is specified in the <forms> element:

```
<!--The restricted folder is for authenticated users and SSL access only-->
<location path="SecureFolderName">
  <system.web>
    <authorization>
      <deny users="?">
    </authorization>
  </system.web>
</location>
```

2.3.2.11.1.3 Create a login Web form

Now when the configuration part is over it's time to build a normal ASP .NET 1.X login form.



The figure above shows how a normal login form may look like. The easiest way to build it is to create a new Web Forms file and drag the following component to it:

- A Label with the Text property set to "User Name:".
- A second Label with the Text property set to "Password:".
- A third Label with the ID property set to "lblMessage".
- A Text Box with the ID property set to "txtUserName".
- A second Text box with the ID property set to "txtPassword" and the TextMode property to "Password".
- A Button with the Text property set to "Register" and the ID property set to "btnRegister".
- A second button Text property set to Logon and the ID property set to "btnLogon".

This procedure creates a simple login page, which allows a user to enter their username and password and submit them to the application.

After that it's time to create the login form's click event handler. You do this by displaying login.aspx in Design view and double click the logon button. You are now in the code behind file login.aspx.cs and the first thing you have to do is to include the System.Web.Security namespace so that you can use the FormsAuthentication class. The next step is to create two private helper methods.

The first helper method you name IsAuthenticated and it will be used to validate the user's name and password against a SQL Server or Active Directory. A simple code example might look like this:

```
private bool IsAuthenticated(string username, string password)
{
    //Here you place the code that authenticates the user against your custom data store.
    return true;
}
```

The second helper method you name `RetrieveRoles` and it will be used to retrieve a user's roles. A simple implementation might look like this:

```
private string RetrieveRoles(string username, string password)
{
    /*
    Here you place the code that retrieves the user's roles. If a user has more than one role
    it's recommended that you return a string that contains pipe separated role names. The
    reason for this is that the string format is convenient for storing in the authentication
    ticket. An example can be seen in the return statement.
    */
    return "Senior Manager|Manager|Employee";
}
```

Now you're ready to program the buttons click event handler. A complete example of the buttons click event handler is presented below:

```
private void Logon_Click(object sender, System.EventArgs e)
{
    //Start by validating the user's submitted credentials against your custom data store
    bool isAuthenticated = IsAuthenticated(txtUsername.Text, txtPassword.Text);

    if(isAuthenticated == true)
    {
        //After that it's time to retrieve the user's roles.
        string roles = RetrieveRoles(txtUsername.Text, txtPassword.Text);
        /*
        When the user's roles are retrieved you create an authentication ticket and store
        the user's roles in the UserData property of the authentication ticket. This improves
        performance by eliminating repeated access to the data store for each Web request
        and also saves you from storing the user's credentials in the authentication cookie.
        */
        FormsAuthenticationTicket authTicket = new FormsAuthenticationTicket(
            1, //version
            txtUserName.Text, //username
            DateTime.Now, //creation
            DateTime.Now.AddMinutes(20), //expiration
            false, //persistent
            roles ); //user data

        /*
        When you have created the authentication ticket you encrypt it. What encryption
        algorithm that will be used is discussed more thoroughly in the Cryptography
        heading, which are found a bit down in this report.
        */
        string encryptedTicket = FormsAuthentication.Encrypt(authTicket);
        /*
        Create a cookie and add the encrypted ticket to the cookie as data. Observe that
        it's recommended to use two cookies; one for secure authentication and
        authorization and one for personalization. The personalization cookie can be made
        persistent, but make sure it doesn't contain any information that would permit a
        request to perform a restricted operation; for example, placing an order within a
        secure part of the site.
        */
        HttpCookie Cookie = new
        HttpCookie(FormsAuthentication.FormsCookieName, encryptedTicket);
        //Add the cookie to the outgoing cookies collection.
        Response.Cookies.Add(authCookie);
        //Redirect the user to the originally requested page.
        Response.Redirect(FormsAuthentication.GetRedirectUrl(txtUserName.Text,
        false));
    }
}
```

As a little reminder, it's recommended that you create a logoff button somewhere on your site, so that the authenticated users can sign out. A typical logoff button click event handler might be implemented like this:

```
void LogoffBtn_Click(object sender, EventArgs e)
{ FormsAuthentication.Signout();}
```

2.3.2.11.1.1.4 Create an IPrincipal object

The IPrincipal object will be created in the Application_AuthenticationRequest event handler, which you find in the file Global.asax. Since we are using Forms authentication we use the GenericPrincipal class to create the IPrincipal object. Observe that you have to add the following two using statements on top of the Global.asax file:

```
using System.Web.Security;
using System.Security.Principal;
```

Now let's take a look at how the Application_AuthenticationRequest event handler might be implemented:

```
protected void Application_AuthenticationRequest(object sender, EventArgs e)
{
    //First you extract the authentication cookie
    string cookiename = FormsAuthentication.FormsCookieName;
    HttpCookie authCookie = Context.Request.Cookies[cookieName];
    if(null == authCookie)
    {
        //There's no authentication cookie
        return;
    }
    FormsAuthenticationTicket authTicket = null;
    try
    {
        authTicket = FormsAuthentication.Decrypt(authCookie.Value);
    }
    catch
    {
        //Log exception details.
        return;
    }
    if(null == authTicket)
    {
        //Cookie failed to decrypt
        return;
    }
    //Retrieve the users roles from the UserData property of the authentication ticket.
    string[] roles = authTicket.UserData.Split(new char[]{'|'});
    //Create an Identity object
    FormsIdentity id = new FormsIdentity(authTicket);
    //Create the GenericPrincipal object that will flow throughout the request.
    GenericPrincipal principal = new GenericPrincipal(id, roles);
    //Attach the the new principal object to the current HttpContext object.
    Context.User = principal;
}
```

The reason why you create a principal object is to give you the possibility to authorize users by their name and / or role(s). There is three ways to do this; declaratively, imperatively and programmatically.

- **Explicit name and role checks.** This is the programmatic way and it may look something like this:

```
//This if statement authorizes the user Johan to perform an operation
if(User.Identity.Name == "Johan"){ }
//or authorize every manager to perform an operation
if(User.IsInRole("Manager")){ }
//or one of them
if(User.Identity.Name == "Johan" || User.IsInRole("Student")){ }
```

```
//or both
```

```
if(User.Identity.Name == "Johan") && User.IsInRole("Student")){ }
```

- **Principal Permission Demands.** There are two types of Principal Permission Demands:
 - **Declarative principal permission.** This you use to restrict access to methods within the application. The following examples show how to use declarative principal permissions:

```
[PrincipalPermissionAttribute(SecurityAction.Demand, User="Johan")]
public void AMethodOnlyForUserJohan(){ }
```

```
//or
```

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Role="Student")]
public void AMethodOnlyForStudents(){ }
```

```
//or both. Observe that it's not possible to perform declarative AND checks
```

```
[PrincipalPermissionAttribute(SecurityAction.Demand, User="Johan"),
PrincipalPermissionAttribute(SecurityAction.Demand, Role="Student")]
public void AMethodOnlyForTheStudentUserJohan(){ }
```

- **Imperative principal permission.** This type you use when to perform fine grained authorization within methods. The following examples show how to use imperative principal permissions:

```
public SomeMethod()
```

```
{
```

```
    PrincipalPermission permCheck = new PrincipalPermission("Johan",
    "Student");
```

```
    permCheck.Demand();
```

```
    /*
```

```
    Only Students with the name Johan can execute the following code. For the
    other users this check will result in a security exception. Observe that null is
    a valid parameter value. If you want a AND operation, for example both
    student and unemployed, you just type two statements as the example
    below shows:
```

```
    */
```

```
    PrincipalPermission permCheck = new PrincipalPermission("Johan",
    "Student");
```

```
    permCheck.Demand();
```

```
    PrincipalPermission permCheck2 = new PrincipalPermission("Johan",
    "Unemployed");
```

```
    permCheckUser2.Demand();
```

```
    /*
```

```
    If you want an OR operation you have to use the PrincipalPermission method
    Union. An example of this are presented below.
```

```
    */
```

```
    PrincipalPermission permCheck = new PrincipalPermission("Johan",
    "Student");
```

```
    PrincipalPermission permCheck2 = new PrincipalPermission("Johan",
    "Unemployed");
```

```
    permCheck.Union(permCheck2).Demand();
```

```
}
```

2.3.2.12 Authentication and authorization countermeasures in ASP .NET 2.0

The authentication and authorization countermeasures have had a major upgrade since ASP .NET 1.X. New features include; seven "login" controls, membership providers and role providers. The membership and role provider provides an interface for the underlying data store(s) through its API functions. If you already have a custom user and role store it's possible to implement custom providers by creating classes that inherits from the MembershipProvider class and RoleProvider class respectively.

To simplify the presentation of the new features we'll create a new web site that uses forms authentication together with the SQL Server membership provider, SqlMembershipProvider, and the SQL Server role provider, SqlRoleProvider. There are other providers available by default, but these are intended for Windows authenticated intranets and extranets only.

Start by open Visual Studio 2005 and create a new ASP .NET Web site. After that you create a new folder called SSLFolder and to that you add a new Web form named Login.aspx.

Now its time to add some of the "login" controls to our page(s). The seven available controls are:

- **Login control.** By default this control has two textboxes (for user name and password), a "Remember Me" checkbox and a button to validate credentials. It's also possible to add links to click if the user has forgotten his password or needs to create a new account. The Login control also provides the ability to validate the user against the default membership provider. If you want to use a different membership provider, than the default, use the MembershipProvider property of the Login control. If you want to customize the authentication process even further it's possible to override the three events; LoggingIn, Authenticate and LoggedIn, which the Login control raises.
- **LoginName control.** Internally this control builds a dynamic instance of a Label control, sets fonts and color accordingly and displays, to the Web page, the text returned by User.Identity.Name property.
- **LoginStatus control.** This control's UI consists of a link button to log in or log out, depending on the current user logon state. If the user is acting as an anonymous user the control displays a link button to invite the user to log in. Otherwise, if the user has already been successfully authenticated, the control displays the Logout button.
- **LoginView control.** The LoginView control lets you aggregate the LoginStatus and LoginName controls allowing you to display a custom UI based on the authentication state and role of the user. The control, which is based on templates, simplifies creation of different UIs that are specific to different states (anonymous or authenticated) and roles.
- **ChangePassword control.** This provides an out of the box, virtually codeless solution that enables end users to change their password. The control supplies a customizable user interface and built in behaviors to retrieve an old password and save a new one. The underlying API for password management is the API supplied by the selected membership provider. The ChangePassword control will work in scenarios where a user may or may not be already authenticated. The control detects if a user is authenticated and automatically populates a user name textbox with the name. Even though the user is authenticated, the user will still be required to reenter the current password. Once the password has been successfully changed, the control may send a confirmation email to the user.
- **PasswordRecovery control.** This control represents the form that enables a user to recover or reset a lost password and receive it via email. After you drop the PasswordRecovery control onto a Web Form, you may have to make some changes to the membership environment before it will work, but only if you want password recovery to use GetPassword in order to send the password back to the user. PasswordRecovery will automatically change its behavior if it's configured as enablePasswordReset=true and enablePasswordRetrieval=false (the default out of the box). In this case, assuming requiresQuestionAndAnswer is set to true, the control will challenge the user with a question and answer. If the answer matches the one that was previously stored, the control will call ResetPassword. At this point, the control can either display the newly reset password or email the password to the user. Note that if the stored password is hashed, retrieving it is completely impossible. In this case, the password can only be reset.
- **CreateUserWizard control.** This is designed to provide native functionality for creating a new user using the standard Membership API. The control offers a basic behavior that the developer can extend to send a confirmation email to the new user and add steps to the wizard to collect additional information, such as address, phone number and roles.

To simplify the example we'll only use the Login and CreateUserWizard controls. Add these to our Login.aspx page and set the DisplayRememberMe property, of the Login control, to false. This will decrease the chance of an attacker stealing the client authentication cookie.

Note that when the user clicks the login button, the Login control automatically validates the user by calling the configured provider (the default provider, if the MembershipProvider isn't set), creates a forms authentication ticket and redirects the user back to the originally requested page.

The code executed by the Authenticate event is similar to the following:

```
if(Membership.ValidateUser(username, password))
{ FormsAuthentication.RedirectFromLoginPage(username, rememberMeIsChecked)}
```

The `Membership.ValidateUser` method validates the user against the specified membership provider's user store and the `FormsAuthentication.RedirectFromLoginPage` method creates an authentication ticket and redirects the user back to the originally requested page.

To the `CreateUserWizard` control we'll only make one change to and that's to the `CreatedUser` event, which is raised after that the membership provider, specified in the `MembershipProvider` property, has created a new user.

To change the event handler of the `CreatedUser` event you start by double clicking the `CreatedUserWizard` control's Properties sheet and this will appear in the `Login.aspx.cs` file:

```
protected void CreateUserWizard1_CreatedUser(object sender, EventArgs e)
{
    if(!Roles.IsUserInRole(CreateUserWizard1.UserName, "LoggedInUserRole"))
        Roles.AddUserToRole(CreateUserWizard1.UserName,
            "LoggedInUserRole");
}
```

The only thing this event handler does, is to check whether the newly created user already is added to the `LoggedInUserRole` role, which we do with the `Roles.IsUserInRole` method, and if not we add the user to the `LoggedInUserRole` role using the `Roles.AddUserToRole` method. The check shouldn't be necessary since a duplicate user can't be created.

Note that the `CreateUserWizard1` is the name of the `CreateUserWizard` object created by default.

After that it's time to configure your application for forms authentication. Add the following to your Web application's `web.config` file:

```
<authentication mode="Forms">
  <forms      loginUrl="RestrictedFolder\MyLoginPage.aspx"
              protection="All"
              timeout="30"
              name="NameOnAuthCookie"
              path="/ApplicationName"
              requireSSL="true"
              slidingExpiration="true"
              cookieless="UseCookies"
              defaultUrl="default.aspx"
              domain="MyWebSite.com"
              enableCrossAppRedirects="false"
            />
</authentication>
```

As you see are there a couple of new attributes, presented in italic, compared to ASP .NET 1.X.

The first new attribute is `cookieless` and it defines whether HTTP cookies are used or not. The following four values are possible:

- **Autodetect.** This value uses cookies if the client browser has cookie support enabled and the `cookieless` mechanism otherwise.
- **UseCookies.** This value forces the `FormsAuthenticationModule` module to always use cookies, regardless of the client browser capabilities.
- **UseDeviceProfile.** This value uses cookies if the browser supports them and the `cookieless` mechanism otherwise. Observe that when this value is used, is no attempt made to check whether cookie support is really enabled.
- **UseUri.** This value forces the `FormsAuthenticationModule` module to never use cookies, regardless of the browser capabilities.

If the `cookieless` mechanism is chosen, are the data, which cookies normally contain, packed into a parenthesized section of the URL, like in the following code snippet:

```
http://localhost/SampleApp/(Ac....Nz)/default.aspx
```

When a request arrives the parenthesized section is stripped off by a module in the HTTP pipeline, so if you read the Request.Path property from an .aspx page, you won't see the parenthesized section of the URL.

The cookieless attribute isn't something the developer will notice since the ASP .NET pipeline will take care of everything that concerns cookieless forms authentication.

The second new attribute is defaultUrl and it lets you set the default name of the page to return after a request has been successfully authenticated. This URL is in ASP.NET 1.X hardcoded to default.aspx but in ASP .NET 2.0 it's configurable. For backward compatibility is the default value default.aspx. Observe that the defaultUrl attribute is used only if no returnUrl variable is found in the URL to the login page. If a user is redirected to the login page by the authentication module, the returnUrl is always correctly set. However, if your page contains a link to the login page or if it needs to transfer programmatically to the login page, you must specify the returnUrl variable and the defaultUrl attribute can, in these "situations", be quite handy.

The third new attribute is domain. This domain attribute is optional and has no default value and is thereby ignored if not explicitly set. If specified it will set the domain property of the HttpCookie class for outgoing authentication cookies. The default value of the HttpCookie.Domain property is the current domain. Observe that this attribute takes precedence over the domain that is used in the httpCookies element and that this attribute is ignored if cookieless authentication is used. This attribute is useful because it lets you share authentication cookies between two or more machines in the same domain. As an example, suppose you have two Web sites named www.MyWebSite.com and weblogs.MyWebSite.com. If you set MyWebSite.com as the authentication domain, the two applications will recognize each other's cookies.

The fourth and final new attribute is enableCrossAppRedirects. It specifies whether authenticated users can or cannot be redirected to URLs in other Web applications. The default value is false.

After that you add the following <authorization> element to your application's Web.config file:

```
<location path="RestrictedFolder">
  <system.web>
    <authorization>
      <deny users="?">
    </authorization>
  </system.web>
</location>
```

There's nothing new here so let's move on. The next step is to configure the membership and role provider. We start with the membership provider. Add the following to your web.config file:

```
<membership defaultProvider="SqlProvider">
  <providers>
    <add name="MySqlProvider"
      type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="MySQLServer"
      enablePasswordRetrieval="false"
      enablePasswordReset="true"
      requiresQuestionAndAnswer="true"
      passwordFormat="Hashed"/>
  </providers>
</membership>
```

This is just some of the attributes that you can use. For a complete list, visit the msdn2 Web site and search for the <membership> element. Observe that the connectionStringName, name and type attributes are required. The connectionStringName attribute specifies the name of a connection string that is defined in the <connectionStrings> element, which the provider uses to connect to your user store. The name attribute specifies the name of the provider instance and must be unique. The type attribute specifies the type that is implementing the MembershipProvider abstract base class.

After that it's time to configure the role provider. Observe that both the membership and role provider creates their own cookies. The membership provider uses the attributes of the authentication element to secure the authentication cookie, while the role manager uses the attributes of the roleManager element to secure the roles cookie. Add the following to the Web.config file to secure the role cookie in the same way as the authentication cookie:

```

<roleManager      enabled="true"
                  defaultProvider="SqlRoleManager"
                  cacheRolesInCookie="true"
                  cookieName=".ASPXROLES"
                  cookiePath="/"
                  cookieProtection="All"
                  cookieRequireSSL="true"
                  cookieSlidingExpiration="true"
                  cookieTimeout="20"
                  createPersistentCookie="false"
                  domain="MyHomePage.com"
                  maxCachedResults="4"
                >
  <providers>
    <add name="SqlRoleManager"
        type="System.Web.Security.SqlRoleProvider"
        connectionStringName="MySQLServer"
        applicationName="MyApplication"
      />
  </providers>
</roleManager>

```

The attributes in the roleManager element are more or less similar to the settings made in the authentication element. The difference is that you have to enable the role manager, specify if and how many roles that should be cached in the role cookie.

The last and final thing you need to do is to create your data source. This can be done in a couple of different ways. All ways are based on the aspnet_regsql.exe tool found in the following directory:

\\WINDOWS\\Microsoft.NET\\Framework\\v2.0.50215

The first option is to use some commands in the command prompt. But the simplest way is to just run the aspnet_regsql.exe from the command prompt or from the above mentioned folder and a Windows application is launched and you create the database in a couple of simple steps.

Observe that the connectionStrings element in your Web.config file should point to this new database. An example of how the connectionStrings element may look like, if your database name is TestDB, is presented below:

```

<connectionStrings>
  <add name="LocalSqlServer" connectionString="data source=(local);Integrated
    Security=SSPI;Database=TestDB"/>
</connectionStrings>

```

2.3.3 Cryptography

Cryptography is all about protecting your data, so that it remains private and unaltered.

The top threats that exploit cryptography vulnerabilities are:

- Checksum spoofing
- Poor key generation or key management
- Weak or custom encryption

2.3.3.1 Threat: Checksum spoofing

Do not rely on hashes to provide data integrity for messages sent over networks. Hashes such as Secure Hash Algorithm (SHA1) and Message Digest compression algorithm (MD5) can be intercepted and changed. Consider the following base 64 encoding UTF-8 message with an appended Message Authentication Code (MAC).

Plaintext: Place 10 orders.

Hash: T0mUNdEQh13IO9oTcaP4FYDX6pU=

If an attacker intercepts the message by monitoring the network, the attacker could update the message and recompute the hash (guessing the algorithm that you used). For example, the message could be changed to:

Plaintext: Place 100 orders.

Hash: oEDuJpv/ZtIU7BXDDNv17EAHeAU=

When recipients process the message and they run the plaintext ("Place 100 orders") through the hashing algorithm and then recompute the hash, the hash they calculate will be equal to whatever the attacker computed.

To counter this attack, use a MAC or HMAC. The Message Authentication Code Triple Data Encryption Standard (MACTripleDES) algorithm computes a MAC and HMACSHA1 computes an HMAC. Both use a key to produce a checksum. With these algorithms, an attacker needs to know the key to generate a checksum that would compute correctly at the receiver.

2.3.3.2 Threat: Poor key generation or key management

Attackers can decrypt encrypted data if they have access to the encryption key or can derive the encryption key. Attackers can discover a key if keys are managed poorly or if they were generated in a non random fashion.

Countermeasures to address the threat of poor key generation and key management include:

- Use built in encryption routines that include secure key management. Data Protection application programming interface (DPAPI) is an example of an encryption service provided on Windows 2000 and later operating systems where the operating system manages the key.
- Use strong random key generation functions and store the key in a restricted location, like in a registry key secured with a restricted ACL, if you use an encryption mechanism that requires you to generate and / or manage the key.
- Encrypt the encryption key using DPAPI for added security.
- Expire keys regularly.

2.3.3.3 Threat: Weak or custom encryption

An encryption algorithm provides no security if the encryption is cracked or is vulnerable to brute force cracking. Custom algorithms are therefore particularly vulnerable if they haven't been tested properly. Instead, use published, well known encryption algorithms that have withstood years of rigorous attacks and scrutiny.

Countermeasures that address the vulnerabilities of weak or custom encryption include:

- Don't develop your own custom algorithms.
- Use the proven cryptographic services provided by the platform.
- Stay informed about cracked algorithms and the techniques used to crack them.

2.3.3.4 Design guidelines for preventing cryptography attacks

Cryptography in its fundamental form provides the following:

- **Privacy** (Confidentiality). This service keeps a secret confidential.
- **Non repudiation** (Authenticity). This service makes sure a user cannot deny sending a particular message.

- **Tamperproofing** (Integrity). This service prevents data from being altered.
- **Authentication**. This service confirms the identity of the sender of a message.

Web applications frequently use cryptography to secure data in persistent stores or as it's transmitted across networks. The following practices improve your Web application's security when you use cryptography (Improving Web application security: Threats and Countermeasures):

- **Do not develop your own cryptography**. Cryptographic algorithms and routines are notoriously difficult to develop successfully. As a result, you should use the tried and tested cryptographic services provided by the platform. This includes the .NET Framework and the underlying operating system. Do not develop custom implementations because these frequently result in weak protection.
- **Keep unencrypted data close to the algorithm**. When passing plaintext to an algorithm, do not obtain the data until you are ready to use it and store it in as few variables as possible.
- **Use the correct algorithm and correct key size**. It's important to make sure you choose the right algorithm for the right job and to make sure you use a key size that provides a sufficient degree of security. Larger key sizes generally increase security. The following list summarizes the major algorithms together with the key sizes that each uses:
 - Data Encryption Standard (DES) 64 – bit key (8 bytes)
 - TripleDES (3DES) 128 – bit key or 192 – bit key (16 or 24 bytes)
 - Rijndael (AES) 128 – 256 bit keys (16 – 32 bytes)
 - RSA 384 – 16,384 bit keys (48 – 2,048 bytes)

For large data encryption, use the 3DES symmetric encryption algorithm. For slower and stronger encryption of large data, use AES. To encrypt data that is to be stored for short periods of time, you can consider using a faster but weaker algorithm such as DES. For digital signatures, use Rivest, Shamir and Adleman (RSA) or Digital Signature Algorithm (DSA). For hashing, use the Secure Hash Algorithm (SHA) 1.0. For keyed hashes, use the Hash based Message Authentication Code (HMAC) SHA1.0.

- **Secure your encryption keys**. An encryption key is a secret number used as input to the encryption and decryption processes. For encrypted data to remain secure, the key must be protected. If an attacker compromises the decryption key, your encrypted data is no longer secure.

The following practices help secure your encryption keys:

- Use DPAPI to avoid key management.
- Cycle your keys periodically.
- **Use DPAPI to avoid key management**. As mentioned previously, one of the major advantages of using DPAPI is that the key management issue is handled by the operating system. The key that DPAPI uses is derived from the password that is associated with the process account that calls the DPAPI functions. Use DPAPI to pass the burden of key management to the operating system.
- **Cycle your keys periodically**. Generally, a static secret is more likely to be discovered over time. Questions to keep in mind are: Did you write it down somewhere? Did Bob, the system administrator change position within the company or even worse quite? Don't overuse keys.

2.3.3.5 Cryptography and sensitive data countermeasure in ASP .NET 1.X

The following countermeasures help you to reduce the risk when you are handling sensitive data within your Web application:

- Avoid plain text passwords and connection strings in configuration files.

- Avoid custom plain text connection strings within your ASP .NET applications
- Don't cache sensitive data.
- Don't pass sensitive data from page to page.
- Use cryptography for securing sensitive data.

2.3.3.5.1 Avoid plain text passwords and connections strings in configuration files.

The <processModel> element and the <identity> element contain username and password attributes and the <sessionState> element contains a connection string attribute in both machine.config and Web.config. Avoid storing these attributes in plaintext. Instead, use the aspnet_setreg.exe tool to encrypt and store these attributes in the registry.

A simple example of how to encrypt the username and password attributes in the <identity> element is presented below:

First, run the following command from the command prompt:

```
aspnet_setreg.exe -k:Software\YourApp\identity -u:"MyUserName" -p:"MyPassword"
```

This command encrypts the userName and password attributes, creates registry keys at the location that you specify and then stores the attributes in those registry keys. This command also generates output that specifies how to change your Web.config or machine.config file, so that ASP .NET will use these keys to read that information from the registry.

Please edit your configuration to contain the following:

```
userName="registry:HKLM\SOFTWARE\YourApp\identity\ASPNET_SETREG,userName"  
password="registry:HKLM\SOFTWARE\YourApp\identity\ASPNET_SETREG,password"
```

The DACL on the registry key grants Full Control to System, Administrators and Creator Owner.

If you have encrypted credentials for the <identity/> configuration section or a connection string for the <sessionState/> configuration section, ensure that the process identity has Read access to the registry key. Furthermore, if you have configured IIS to access content on a UNC share, the account used to access the share will need Read access to the registry key. Regedt32.exe may be used to view / modify registry key permissions.

You may rename the registry subkey and registry value in order to prevent discovery.

The aspnet_setreg.exe utility uses the Data Protection API (DPAPI), which is provided by the OS, to encrypt and decrypt the username and password. Observe the registry: prefix in the output generated by the command. The registry: prefix tells ASP .NET worker process to get the real value from the registry and decrypt it.

The DPAPI consists of two methods (CryptProtectData and CryptUnprotectData) and it's particularly useful since it eliminates the key management problems exposed to applications that use cryptography. The aspnet_setreg.exe utility sets the CRYPTPROTECT_LOCAL_MACHINE flag on the CryptProtectData function, which means that any user on the local computer can call the CryptUnprotectData function to decrypt the message. Because anyone with access to the computer can call CryptUnprotectData, the encrypted data is stored under a secure registry key with a strong discretionary access control list (DACL).

By default, the registry keys that Aspnet_setreg.exe creates grant full control to the System, Administrator and Creator Owner accounts. You can use Regedt32.exe to modify the DACL on the registry key. The last thing you need to do, to make this work, is to grant either the ASPNET or the NetWork Service account, depending on which Windows version your Web server runs under, read permissions. You do this in the following way:

1. Click Start, click Run, type regedt32 in the Open box and then click OK.
2. Click the HKEY_LOCAL_MACHINE\SOFTWARE\YourApp\subkey.

3. On the Security menu, click Permissions to open the Permissions dialog box. On Microsoft Windows XP or on Windows Server 2003, right click the registry key and then click Permissions.
4. Click Add. In the dialog box that opens, type yourservername\ASPNET (or yourservername\NetWork Service when using Windows Server 2003 (IIS 6.0)) and then click OK.
5. Make sure that the account that you just added has Read permissions and then click OK.
6. Close Registry Editor

It's also very important that only the accounts that your application needs, gets read permissions to these registry keys.

The same procedure applies when encrypting the connection string in the <sessionState> element. The only difference is the command, which you switch to the following:

aspnet_setreg -k:Software\YourApp\sessionState -c:"My connection string"

Don't forget to edit the sqlConnectionString attribute in the <sessionState> element.

sqlConnectionString="registry:HKLM\SOFTWARE\YourApp\SessionState\ASPNET_SET REG,sqlConnectionString"

The drawback with this approach is when your deployment environment is a Web farm. This is though, still the recommended approach even when using Web farms. Security is all about trade offs and in this case, the benefit of greater security comes at the cost of administrative overhead at deployment of ASP .NET 1.X applications.

2.3.3.5.2 Avoid custom plain text connection strings within your ASP .NET applications

If you create a custom data store, which requires a custom connection string, it's recommended that you store the connection string in the registry in an encrypted form. The basic principle is to use the Encryptor and Decryptor classes in the Encryption namespace. A good example is presented on page 440 in Building Secure ASP.NET applications.

2.3.3.5.3 Don't cache sensitive data

If your page contains sensitive data, such as passwords, credit cards number or accounts status, it shouldn't be cached. By default are the output caching turned off in ASP .NET.

2.3.3.5.4 Don't pass sensitive data from page to page

Avoid using any of the client side state management options, such as view state, cookies, query strings or hidden form field variables to store sensitive data. The data can be tampered with and viewed in clear text. Use server side state management options, such as a SQL Server database for secure data exchange.

2.3.3.5.5 Use cryptography for securing sensitive data

Cryptography is one of the most important tools for protecting sensitive data. You should use encryption when you want data to be secure in transit or in storage. Some encryption algorithms perform better than others while some provide stronger encryption. Typically, larger encryption key sizes increase security.

To make sure that your ASP .NET Web application is secure, it's important that you understand and use the following guidelines:

- Use platform provided cryptographic services
- Key generation
- Key storage

- Key exchange
- Key maintenance

2.3.3.5.5.1 Use platform provided cryptographic services

Don't create your own cryptographic implementations. It's extremely unlikely that these implementations will be as secure as the industry standard algorithms provided by the OS and the .NET Framework. Managed code should use the algorithms provided by the System.Security.Cryptography namespace for encryption, decryption, hashing, random number generating and digital signatures.

Many of the types in this namespace wrap the operating system CryptoAPI, while others implement algorithms in managed code.

2.3.3.5.5.2 Key generation

The following recommendations apply when you create encryption keys:

- Generate random keys.
- Prefer large keys.

2.3.3.5.5.2.1 Generate random keys

If you need to generate encryption keys programmatically, use RNGCryptoServiceProvider for creating keys and initialization vectors and do not use the Random class. Unlike the Random class, RNGCryptoServiceProvider creates cryptographically strong random numbers which are FIPS 140 compliant. The following code shows how to use this function.

```
using System.Security.Cryptography;
...
...
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
byte[] key = new byte[keySize];
rng.GetBytes(key);
```

2.3.3.5.5.2.2 Prefer large keys

When generating an encryption key or key pair, it's recommended that you use the largest possible key size. This does not necessarily make the algorithm more secure but dramatically increases the time needed to successfully perform a brute force attack on the key. The following code shows how to programmatically find the largest supported key size for a particular algorithm:

```
private int GetLargestSymKeySize(SymmetricAlgorithm symAlg)
{
    KeySizes[] sizes = symAlg.LegalKeySizes;
    return sizes[sizes.Length].MaxSize;
}

private int GetLargestAsymKeySize(AsymmetricAlgorithm asymAlg)
{
    KeySizes[] sizes = asymAlg.LegalKeySizes;
    return sizes[sizes.Length].MaxSize;
}
```

The symmetric algorithms implemented in .NET Framework are: DESCryptoServiceProvider, RC2CryptoServiceProvider, RijndaelManaged and TripleDESCryptoServiceProvider.

The asymmetric algorithms implemented in the .NET Framework are: RSACryptoServiceProvider and DSACryptoServiceProvider.

2.3.3.5.5.3 Key storage

Where possible, you should use a platform provided encryption solution that enables you to avoid key management in your application. However, at times you need to use encryption solutions that require you to store keys. Using a secure location to store the key is critical. Use the following techniques to help prevent key storage vulnerabilities:

- Use DPAPI to avoid key management.
- Don't store keys in code.
- Restrict access to persisted keys.

2.3.3.5.5.3.1 Use DPAPI to avoid key management

DPAPI is a native encryption / decryption feature provided by Windows 2000 and later. One of the main advantages of using DPAPI is that the encryption key is managed by the operating system, because the key is derived from the password that is associated with the process account (or thread account if the thread is impersonating) that calls the DPAPI functions.

You can perform encryption with DPAPI using either the user key or the machine key. By default, DPAPI uses a user key. This means that only a thread that runs under the security context of the user account that encrypted the data can decrypt the data. You can instruct DPAPI to use the machine key by passing the `CRYPTPROTECT_LOCAL_MACHINE` flag to the `CryptProtectData` function. In this event, any user on the current computer can decrypt the data.

The user key option can be used only if the account used to perform the encryption has a loaded user profile. If you run code in an environment where the user profile is not loaded, you cannot easily use the user store and should opt for the machine store instead.

Version 1.1 of the .NET Framework loads the user profile for the ASPNET account used to run Web applications on Windows 2000. Version 1.0 of the .NET Framework does not load the profile for this account, which makes using DPAPI with the user key more difficult.

If you use the machine key option, you should use an ACL to secure the encrypted data, for example in a registry key, and use this approach to limit which users have access to the encrypted data. For added security, you should also pass an optional entropy value to the DPAPI functions.

The drawback with using entropy is that you must manage the entropy value as you would manage a key. To avoid entropy management issues, use the machine store without entropy and validate users and code (using code access security) thoroughly before calling the DPAPI code.

2.3.3.5.5.3.2 Don't store keys in code

Don't store keys in code because hard coded keys in your compiled assembly can be disassembled using tools similar to ILDASM, which will render your key in plaintext.

2.3.3.5.5.3.3 Restrict access to persisted keys

When storing keys in persistent storage to be used at runtime, use appropriate ACLs and limit access to the key. Access to the key should be granted only to Administrators, SYSTEM and the identity of the code at runtime, for example the ASPNET or NetWork Service account.

When backing up a key, don't store it in plain text, encrypt it using DPAPI or a strong password and place it on removable media.

2.3.3.5.5.4 Key exchange

Some applications require the secure exchange of encryption keys over an insecure network. You may need to verbally communicate the key or send it through secure e-mail. A more secure method to exchange a symmetric key is to use public key encryption. With this approach, you encrypt the symmetric key to be exchanged by using the other party's public key from a certificate that can be validated. A certificate is considered valid when:

- It's being used within the date ranges as specified in the certificate.
- All signatures in the certificate chain can be verified.
- It's of the correct type. For example, an e-mail certificate is not being used as a Web server certificate.
- It can be verified up to a trusted root authority.
- It's not on a Certificate Revocation List (CRL) of the issuer.

2.3.3.5.5.5 Key maintenance

Security is dependent upon keeping the key secure over a prolonged period of time. One way to keep the key secure is to cycle it periodically. You do this because a static secret is more likely to be discovered over time. Did you write it down somewhere? Did Bob the administrator with the secrets change positions in your company or leave the company? Are you using the same session key to encrypt communication for a long time? Do not overuse keys.

2.3.3.5.6 Key compromise

Keys can be compromised in a number of ways. For example, you may lose the key or discover that an attacker has stolen or discovered the key.

If your private key used for asymmetric encryption and key exchange is compromised, don't continue to use it and notify the users of the public key that the key has been compromised. If you used the key to sign documents, they need to be resigned.

If the private key of your certificate is compromised, contact the issuing certification authority to have your certificate placed on a certificate revocation list. Also, change the way your keys are stored to avoid a future compromise.

2.3.3.5.6.1 How to store sensitive data in a database

Many Web applications store sensitive data of one form or another in the database. If an attacker manages to execute a query against your database, it's extremely important that any sensitive data items, such as credit card numbers, are suitably encrypted.

The first thing you have to ask yourself, when you want to store sensitive data, is whether you really need to store it or not. If you must store the data, use the following guidelines to encrypt and decrypt it.

In this example we'll use 3DES to encrypt the data.

During development, the first step is to use the `RNGCryptoServiceProvider` class to generate a strong (192 bit, 24 byte) encryption key. An example of how to do this is presented below:

```
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();  
rng.GetBytes(24);
```

Then it's time to back up the encryption key and store the backup in a physically secure location.

After that you encrypt the key with DPAPI and store it in a registry key. Use the following ACL to secure the registry key:

```
Administrators: Full Control  
Process Account (for example ASPNET): Read
```

At runtime you use the following steps to encrypted the data and store it in the database:

- Obtain the data to be encrypted.
- Retrieve the encrypted encryption key from the registry.

- Use DPAPI to decrypt the encryption key.
- Use the TripleDESCryptoServiceProvider class with the encryption key to encrypt the data.
- Store the encrypted data in the database.

At runtime you use the following steps to decrypt the data stored in the database:

- Retrieve the encrypted data from the database.
- Retrieve the encrypted encryption key from the registry.
- Use DPAPI to decrypt the encryption key.
- Use the TripleDESCryptoServiceProvider class to decrypt the data.

With this process, if the DPAPI account used to encrypt the encryption key is damaged, the backup of the 3DES key can be retrieved from the backup location and be encrypted using DPAPI under a new account. The new encrypted key can be stored in the registry and the data in the database can still be decrypted.

2.3.3.5.6.2 Password storage

Another common Web application task is to store password hashes with salt in the database.

If you need to implement a user store that contains user names and passwords, don't store the passwords either in clear text or in encrypted format. Instead of storing passwords, store non reversible hash values with added salt to mitigate the risk of dictionary attacks. A salt value is a cryptographically strong random number.

The following code shows how to generate a salt value by using random number generation functionality provided by the RNGCryptoServiceProvider class.

```
public static string CreateSalt(int size)
{
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
    byte[] buff = new byte[size];
    rng.GetBytes(buff);
    return Convert.ToBase64String(buff);
}
```

After you have generated the salt you can generate a hash value from the supplied password and the salt value. An example of how to do this is presented below:

```
public static string CreatePasswordHash(string pwd, string salt)
{
    string saltAndPwd = string.Concat(pwd, salt);
    string hashedPwd = FormsAuthentication.HashPasswordForStoringInConfigFile(
        saltAndPwd, "SHA1");
    return hashedPwd;
}
```

2.3.3.5.6.3 The machineKey element

The <machineKey> element is used to specify encryption keys, validation keys and algorithms that are used to protect Forms authentication cookies and page level view state. The <machineKey> element contains the following three sub elements in ASP .NET 1.X:

- **decryptionKey**. This attribute specifies the key used to encrypt data. The decryptionKey attribute is used for Forms authentication encryption (FormsAuthentication.Encrypt) and decryption (FormsAuthentication.Decrypt) and for view state encryption when the validation attribute is set to "3DES".

This attribute can have one of the following two values:

- **AutoGenerate.** This is the default value and specifies that ASP .NET generates a random key and stores it in the Local Security Authority (LSA).
- The second value specifies a manually assigned encryption key, which is a must when you want to ensure consistent configuration across Web farms. The key should be a minimum of 8 bytes (16 characters) long when using DES encryption and 24 bytes (48 characters) long when using Triple DES encryption. If you decide to use a shorter key, it's recommended that you use the RNGCryptoServiceProvider to create a truly random value. Observe that ASP .NET only can use Triple DES on computers that have 128 bit encryption "enabled". This is "enabled" on all Windows Systems since Windows 2000 SP2.

If you intend to deploy multiple applications on a Web server or in a Web farm, which don't need to share authentication tickets across two or more Web applications, you have to apply the correct key setting to ensure application isolation. The following two settings are available:

- The first setting is applicable to single server deployments. You apply this setting by adding the `IsolateApps` modifier to the `validationKey` value, as shown below:

validationKey = "AutoGenerate|value,IsolateApps"

By doing this ASP .NET generates a unique encrypted key for each Web application using each Web application's application ID. Observe that this modifier isn't available in ASP .NET 1.0. Suggestions of how to solve this issue, can be found at Microsoft Help and Support; Microsoft Knowledge Base article 313116.

- The second setting mainly applicable to Web farm deployments, but it can also be used for single server deployments. You apply this setting by placing the `<machineKey>` element in the `Web.config` file for each application on each server in a Web farm. Ensure that you use separate key values for each application, but duplicate each application's keys across all servers in the Web farm.
- **validation.** This attribute specifies the hashing algorithm used to generate HMACs to make forms authentication tickets and ViewState tamper proof. This attribute is also used to specify the encryption algorithm used for ViewState encryption.

The possible values for this attribute are:

- **3DES.** If you set `validation="3DES"` on the `<machineKey>`, then page level view state is encrypted, which also provides integrity checking, using the 3DES algorithm, even if the `<pages>` element is configured for view state MACs. Observe that when 3DES is specified, forms authentication defaults to SHA1.

If you used `protection="All"` on the `<forms>` element, then the Forms authentication ticket is encrypted, which also ensures integrity. Regardless of the validation attribute, Forms authentication uses TripleDES (3DES) to encrypt the ticket and it isn't configurable.

Note that Forms authentication cookie encryption is independent of the `validationkey` setting and the key is based on the `decryptionKey` attribute.

- **MD5.** You can set the validation attribute to MD5, but you should use SHA1 since that algorithm produces a larger hash than MD5 and is therefore considered cryptographically stronger.
- **SHA1.** If you set `validation="SHA1"` on the `<machineKey>`, then page level view state is integrity checked using the SHA1 algorithm if the `<pages>` element's `enableViewStateMac` attribute is set to `true`, which is the default value. The algorithm used is HMACSHA1.
- **validationKey.** This attribute specifies the key used for validation of data. Examples of when this key is used is when `enableViewStateMAC` is set to `"true"` to create a message authentication code (MAC) to ensure that view state isn't tampered with and to generate out of process application specific session IDs to ensure that session state variables are isolated between sessions.

This attribute can have one of the following two values:

- **AutoGenerate.** This is the default value and specifies that ASP .NET generates a random key and stores it in the Local Security Authority (LSA).
- The second value specifies a manually assigned validation key, which is a must when you want to ensure consistent configuration across Web farms. The key must be a minimum of 20 bytes (40 characters) and a maximum of 64 bytes (128 characters) long. The recommended key length is 64 bytes, but if you decide to use a shorter key, it's recommended that you use the RNGCryptoServiceProvider to create a truly random value.

If you intend to deploy multiple applications on a Web server or in a Web farm, the same attribute settings as the decryptionKey attribute applies.

2.3.3.6 Cryptography and sensitive data countermeasure in ASP .NET 2.0

The first big change to the cryptography and sensitive data countermeasures in ASP .NET 2.0, are the introduction of the decryption attribute in the <machineKey> element. The attributes in the <machineKey> element are, in ASP .NET 2.0, the following:

- **decryption.** The decryption attribute specifies the symmetric encryption algorithm used to encrypt and decrypt forms authentication tickets. The algorithms that available are; 3DES and AES. AES is new to the <machineKey> element in ASP .NET 2.0 and you should use it since it provides larger key sizes (128, 192, and 256) than 3DES.

The default value of the decryption attribute is "Auto". If the value of the decryptionKey attribute is 8 bytes long (16 characters) then Auto defaults to DES. Otherwise, Auto defaults to AES. Supported values include 3DES, AES, Auto and DES.

- **decryptionKey.** The only new "feature" for this attribute is that you can set the key to a 32 byte (64 characters) value instead of a 24 byte (48 characters), which was the maximum in ASP.NET 1.X. This key is used for encrypting and decrypting authentication tickets, role cookies and anonymous identification cookies. ASP .NET does also use this key to encrypt and decrypt ViewState, but only if validation attribute is set to 3DES or AES.
- **validation.** The validation attribute do, in ASP .NET 2.0, have support for a new value; AES. The AES value has the exact functionality as the 3DES value, meaning that the AES value specifies that the AES algorithm should be used when encrypting the ViewState and the key should be the one specified in the validationKey attribute. When AES is chosen, forms authentication ticket and ViewState are tamperproofed using the SHA1 algorithm and the validationKey value, like the 3DES value do in both ASP .NET 1.X and 2.0.
- **validationKey.** This attribute is, like in ASP .NET 1.X, used for signing the authentication ticket and tamper proofing the ViewState. The news is that the Role Manager and anonymous identification, if any of these are enabled, sign their cookies with this attribute value. If you use anonymous identification in cookieless mode, the data on the URL is also signed with this attribute value.

The second big change is the possibility to encrypt the sections of your configuration files that contains sensitive information, such as connection strings and passwords.

There are two different ways of doing this in ASP .NET 2.0 and they are the following:

- The first way is to use the DataProtectionConfigurationProvider, which uses the DPAPI to encrypt and decrypt sensitive data, together with the aspnet_regiis.exe tool.
- The second way is to use the RSAProtectedConfigurationProvider, which uses RSA public key encryption to encrypt and decrypt sensitive data, together with the aspnet_regiis.exe tool. The RSAProtectedConfigurationProvider is the default provider and is the option to choose if you're planning to use a Web farm, since RSA keys very easily can be exported to other computers.

You use these protected configuration providers in the following way:

- Step one is to identify the section to be encrypted. The following configuration sections can't be encrypted by any of the protected configuration providers:
 - <processmodel>

- <runtime>
- <mscorlib>
- <startup>
- <system.runtime.remoting>
- <protectedData>
- <satelliteassemblies>
- <cryptographySettings>
- <cryptoNameMapping>
- <cryptoclasses>

Instead you've to use the `aspnet_setreg.exe` tool to encrypt these sections.

For the upcoming example we'll choose to encrypt the <connectionStrings> section.

- Step two is to choose whether to use DPAPI's machine or user store or RSA's machine or user level key containers.

The DPAPI's machine store and RSA's machine level key container is recommended to use in the following two scenarios:

- When your application runs on its own server with no other application.
- You've multiple applications on the same server and you want these applications to be able to share sensitive information.

DPAPI's user store and RSA's user level key container should be used; if you run your application in a shared hosting environment and you want to make sure that your application's sensitive data isn't accessible to other applications on the same server.

For the upcoming example we choose to use a RSA user level key container.

- Step three is to encrypt the selected section. You do this in the following way:
 - The first step, when using the `RSAProtectedConfigurationProvider`, is to create a new key container and it doesn't matter if you intend to use a machine or user level key container. To do this you open a command prompt and write and execute the following two commands:

```
cd \WINDOWS\Microsoft.Net\Framework\v2.0.*
```

```
aspnet_regiis -pc "MyKeys" -pku -exp
```

With the first command you navigate to the folder of the `aspnet_regiis.exe` tool and with the other command you create the key container. The switches used in the second command are the following:

- **-pc containerName**. This switch creates a RSA key pair in the container `containerName`. In our example are the container name "MyKeys".
- **-pku**. This is an argument to the `-pc` switch and specifies that you want to use a user container instead of a machine container.
- **-exp**. This switch ensures that the key is exportable.
- **-size keysize**. The default key size is 1024 bits. This switch isn't used in this example, but it might be useful to know.

The second command generates the following output if it successfully managed to create the specified the key container:

**Creating RSA Key container...
Succeeded!**

The user level key containers are stored in the following folder:

**\Documents and Settings\{UserName}\Application
Data\Microsoft\Crypto\RSA**

If you instead would have chosen to use a machine key container, it would have been stored in the following folder:

**\Documents and Settings\All Users\Application
Data\Microsoft\Crypto\RSA\MachineKeys**

- After that it's time to add the following <protectedData> section to your Web application's Web.config file:

```
<protectedData><providers>
  <add
    keyContainerName="MyKeys"
    useMachineContainer= "false"
    name="MyRSAProtectedConfigurationProvider"
    type="System.Configuration.RSAProtectedConfiguration
    Provider, System.Configuration, Version=2.0.0.0,
    Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  />
</providers></protectedData>
```

There are two important things you need to keep in mind when configuring the <protectedData> section. The first is to set useMachineContainer="false". This instructs the provider to use a user level key container. The second is to use a unique provider name; otherwise will a runtime error be generated. The default provider name is RSAProtectedConfigurationProvider and can therefore not be used when creating a custom provider. The default RSAProtectedConfigurationProvider has the useMachineContainer attribute set to "true" by default and the RSA key container that it uses is named "NetFrameworkConfigurationKey".

- After that you run the following command from a command prompt:

```
aspnet_regiis -pe "connectionStrings" -app "/WebApplicationName" -
prov "MyRSAProtectedConfigurationProvider"
```

This command encrypts the <connectionStrings> section of your application's Web.config file, using the following switches:

- The **-app** switch specifies your Web application's virtual path. If it's a nested application, you need to specify the nested path from the root directory, as shown in the following example:

```
"/WebApplicationName/test/RSAUserKey"
```

- The **-pe** switch specifies the configuration section to encrypt. This is the XML element name of the configuration section.

For nested elements, such as the <pages> section, which is inside the <system.web> element, must the XML name of the "parent" element be included, as shown in the following example:

```
"system.web/pages"
```

- The **-prov** switch specifies the provider name. In this case is this set to "MyRSAProtectedConfigurationProvider" which is the name we specified earlier in the <protectedData> element. Observe that when using the default RSAProtectedConfigurationProvider with its default settings you don't use this switch.

If the command is successful, you'll see the following output:

**Encrypting configuration section...
Succeeded!**

The result, in your application's web.config file, should look similar to this:

```
<configuration>
  <protectedData>
    <providers>
      <add
        keyContainerName="MyKeys"
        useMachineContainer="false"
        name="MyRSAProtectedConfigurationProvider"
        type="System.Configuration.RSAProtectedConfig
          urationProvider, System.Configuration,
          Version=2.0.0.0, Culture=neutral,
          PublicKeyToken=b03f5f7f11d50a3a"
      />
    </providers>
    <protectedDataSections>
      <add
        name="connectionStrings"
        provider="MyRSAProtectedConfigurationProvider"
      />
    </protectedDataSections>
  </protectedData>
  <connectionStrings>
    <EncryptedData
      Type=http://www.w3.org/2001/04/xmlenc#Element
      xmlns=http://www.w3.org/2001/04/xmlenc#
    >
    <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#trip
        ledes-cbc"/>
    <KeyInfo
      xmlns="http://www.w3.org/2000/09/xmldsig#">
      <EncryptedKey
        Recipient=""
        xmlns="http://www.w3.org/2001/04/xml
          enc#">
        <EncryptionMethod
          Algorithm="http://www.w3.org/200
            1/04/xmlenc#rsa-1_5"/>
        <KeyInfo
          xmlns="http://www.w3.org/2000/0
            9/xmldsig#">
          <KeyName>Rsa Key</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>
            emy+cEGj6qxM6xxwrdoFAhTQ
            Aj9OFi81ZXhnTEKgZQuhxOrfF
            +A7w/7ud99C29ggZ90HwjHRJ
            qgJ6h9U5OmUiaz8WqHj7b4aM
            AHhROCHRWR6KHtEEcfAL8uHP
            2yikP+vEMsYWpEuY0HIJ7SWB
            QW6ES6Du6zhtzDn081A9Z6Yj
            R4=
          </CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>
        okMVI GnJCNctVxOoSgZwygTd8nC9P1Pijkv
        +wIWQE+ij1DIJxPpnNAFrLXadKg2LCyEN4x
        uHtOT+5vxGPisDy/yn+osF1zbC1zpA3FiHvp
      </CipherValue>
    </CipherData>
  </connectionStrings>
</configuration>
```

```

VL/mjN4tZXyNku9zWCs5It1B1880CX0sIDo
4JJHNXhxptIj8F72KsUZegJnxaK7c6+ZBfA4
zzAcTL6uisZIG5JEttQq7Z35nyKErIA95wXlb
NfqggySWo0/RM9VMyy8Hs=
</CipherValue>
</CipherData>
</EncryptedData>
</connectionStrings>
<configuration>

```

If you decide to decrypt the connectionStrings section use the following command:

```
aspnet_regiis -pd "connectionStrings" -app "/WebApplicationName"
```

The decryption command is very similar to the encryption command syntax. The two exceptions are that you replace the `-pe` switch with the `-pd` switch and that you don't specify a protected configuration provider. The reason why you don't specify a protected configuration provider is that the appropriate provider is identified in the `<protectedDataSections>` element in the Web.config file.

- The fourth and final step is to grant your ASP .NET Web application's service account identity, which is ASPNET, on Windows XP Professional, and NT Authority\Network Service, on Windows Server 2003, read access to your key container.

If you've followed the steps above, you've created the key container using your local Windows account, which is the only account that are given read access to your key container. To grant the NT Authority\Network Service process account read access to the key container run the following command:

```
aspnet_regiis -pa "MyKeys" "NT Authority\Network Service"
```

If the command runs successfully, you'll see the following:

```
Adding ACL for access to the RSA Key container...
Succeeded!
```

Without the correct RSA key container used to encrypt the web.config file, ASP .NET will be unable to decrypt the encrypted configuration values. So if you intend to use a Web farm you must export your RSA key container to a XML file and then copy the XML file to the Web server(s) that will host a copy of your Web application. To export your RSA key container to a XML file, run the following command from a command prompt:

```
aspnet_regiis -px "MyKeys" "c:\keys.xml" -pri
```

This command's `-px` switch/option exports the RSA key container named "MyKeys" to the keys.xml XML file in the root directory of the c:\ drive. The `-pri` switch is optional and it specifies that the private key should be exported. If the `-pri` switch is omitted, the exported key will only be able to encrypt information, not decrypt it.

After that you copy the keys.xml file to the root directory of the c:\ drive and run the following command to import the key container:

```
aspnet_regiis -pi "MyKeys" "c:\keys.xml"
```

The `-pi` switch imports the key container named MyKeys to root directory of the c:\ drive.

The final thing needed before you completed the exporting of the RSA key container Mykeys is to grant your ASP .NET Web application's service account identity read access to the key container.

The third big change to the cryptography and sensitive data countermeasures in ASP .NET 2.0, are the two DPAPI wrapper classes; ProtectedData and ProtectedMemory.

The ProtectedData class consists of two wrapper functions for the unmanaged DPAPI; Protect and Unprotect. The Protect function takes the following three parameters:

- The first parameter is a byte array containing the data to protect.

- The second parameter is an optional entropy byte array.
- The third parameter is one of the `DataProtectionScope` values. The possible values are:
 - **`DataProtectionScope.CurrentUser`**. This value is associated with the current user and only threads running under the current user context can unprotect the data when this value is used.
 - **`DataProtectionScope.LocalMachine`**. This value is associated with the machine context and any process running on the computer can unprotect data.

The return value of the `Protect` function is a byte array representing the encrypted data. The `Unprotect` function takes the same parameters as the `Protect` function. Observe that the first parameter is the encrypted byte array and second and third parameters are the same entropy and `DataProtectionScope` value as were used with the `Protect` function.

The `ProtectedMemory` class has the same two wrapper `Protect` and `Unprotect` functions. The only difference is the parameters and the return value, which in the `ProtectedMemory` class doesn't have a return value. Both functions take the byte array that should be encrypted / decrypted as its first parameter and one of the `MemoryProtectionScope` values as its second parameter. The possible `MemoryProtectionScope` values are:

- **`MemoryProtectionScope.CrossProcess`**. All code in any process can unprotect memory that was protected using the `Protect` method.
- **`MemoryProtectionScope.SameLogon`**. Only code running in the same user context as the code that called the `Protect` method can unprotect memory.
- **`MemoryProtectionScope.SameProcess`**. Only code running in the same process as the code that called the `Protect` method can unprotect memory.

2.3.4 Exception management

Exceptions that are allowed to propagate to the client can reveal internal implementation details that make no sense to the end user but are useful to attackers. Applications that do not use exception handling or implement it poorly are also subject to denial of service attacks. Top exception handling threats include:

- Attacker reveals implementation details
- Denial of service

2.3.4.1 Threat: Attacker reveals implementation details

One of the important features of the .NET Framework is that it provides rich exception details that are invaluable to developers. If the same information is allowed to fall into the hands of an attacker, it can greatly help the attacker to exploit potential vulnerabilities and plan future attacks. The type of information that could be returned includes platform versions, server names, SQL command strings and database connection strings.

Countermeasures to help prevent internal implementation details from being revealed to the client include:

- Use exception handling throughout your application's code base.
- Handle and log exceptions that are allowed to propagate to the application boundary.
- Return generic, harmless error messages to the client.

2.3.4.2 Threat: Denial of service (DOS)

Attackers will probe a Web application, usually by passing deliberately malformed input. They often have two goals in mind. The first is to cause exceptions that reveal useful information and

the second is to crash the Web application process. This can occur if exceptions are not properly caught and handled.

Countermeasures to help prevent application level denial of service include:

- Thoroughly validate all input data at the server.
- Use exception handling throughout your application's code base.

2.3.4.3 Design guidelines for preventing exception management attacks

Secure exception handling can help prevent certain application level denial of service attacks and it can also be used to prevent valuable system level information useful to attackers from being returned to the client. For example, without proper exception handling, information such as database schema details, operating system versions, stack traces, file names and path information, SQL query strings and other information of value to an attacker can be returned to the client.

A good approach is to design a centralized exception management and logging solution and consider providing hooks into your exception management system to support instrumentation and centralized monitoring to help system administrators.

The following practices help secure your Web application's exception management:

- **Don't leak information to the client.** In the event of a failure, do not expose information that could lead to information disclosure. For example, do not expose stack trace details that include function names and line numbers in the case of debug builds (which should not be used on production servers). Instead, return generic error messages to the client.
- **Log detailed error messages.** Send detailed error messages to the error log. Make sure though that you don't log passwords or other sensitive data.
- **Catch exceptions.** It's recommended that you use structured exception handling and catch exception conditions. Doing so avoids leaving your application in an inconsistent state that may lead to information disclosure. It also helps protect your application from denial of service attacks. Decide how to propagate exceptions internally in your application and give special consideration to what occurs at the application boundary.

2.3.4.4 Exception management countermeasures in ASP .NET

Correct exception handling in your Web pages prevents sensitive exception details from being revealed to the user. The following recommendations apply to ASP .NET Web pages and controls (Improving Web application security: Threats and Countermeasures):

- **Use structured exception handling.** C# provides the try, catch and finally structured exception handling constructs.

You protect code by placing it inside try blocks and implement catch blocks to log and process exceptions. Also use the finally construct to ensure that critical system resources, such as connections, are closed irrespective of whether an exception condition occurs. The following example shows how to use these constructs:

```
try
{ //Here you place the code that could throw an exception. }
catch
{ //Here you place the code that should handle the exception and log details. }
finally
{ //Here you place the code that ensures that resources are closed or released. }
```

Use structured exception handling instead of returning error codes from methods, because it's easy to forget to check a return code and as a result fail to an insecure mode.

- **Return generic error pages to the client.** In the event of an unhandled exception, return a generic error page to the user. You do this by configuring the `<customErrors>` element, in either `machine.config` or `Web.config`, in the following way:
`<customErrors mode="on" defaultRedirect="YourErrorPage.aspx">`
 The error page should include a suitably generic error message and possibly additional support details.

The name of the page that generated the error is passed to the error page through the `aspxerrorpath` query parameter.

You can also use multiple error pages, as shown in the example below, for different types of errors.

```
<customErrors mode="on" defaultRedirect="YourErrorPage.aspx">
  <error statusCode="404" redirect="YourNotFoundPage.html"/>
  <error statusCode="500" redirect="YourInternalErrorPage.html"/>
</customError>
```

For individual pages you can supply an error page using the following page level attribute:

```
<% @Page ErrorPage="YourErrorPage"%>
```

- **Implement page or application level error handlers.** If you need to trap and process unhandled exceptions at page level, create a handler for the `Page_Error` event that is similar to the one shown in the example below:

```
public void Page_Error(object sender, EventArgs e)
{
    //Get the source exception details
    exception ex = Server.GetLastError();

    /*
    Write the details to the event log
    */

    /*
    Prevent the exception from generating an application level event (Application.Error)
    */
    Server.ClearError();
}
```

If you decide to allow application error events to be raised, from the page error event handler or if no page error event handler is implemented, you can trap these events by implementing an application error event handler, `Application_Error`, in `Global.asax`.

```
protected void Application_Error(object sender, EventArgs e){}
```

2.3.5 Input validation

Input validation is a security issue if an attacker discovers that your application makes unfounded assumptions about the type, length, format or range of input data. The attacker can then supply carefully crafted input that compromises your application.

When network and host level entry points are fully secured, the public interfaces exposed by your application become the only source of attack. The input to your application is a means to both test your system and a way to execute code on an attacker's behalf. If your application blindly trusts input, it may be susceptible to the following types of input validation threats:

- Buffer overflow
- Canonicalization
- SQL injection
- XSS

2.3.5.1 Threat: Buffer overflows

Buffer overflow vulnerabilities can lead to denial of service attacks, which causes a process to crash, or code injection, which alters the program execution address to run an attacker's injected code. The following code fragment illustrates a common example of a buffer overflow vulnerability.

```
void SomeFunction( char *pszInput )
{
    char szBuffer[10];
    // Input is copied straight into the buffer when no type checking is performed
    strcpy(szBuffer, pszInput);
    ...
}
```

Managed .NET code is not susceptible to this problem because array bounds are automatically checked whenever an array is accessed. This makes the threat of buffer overflow attacks on managed code much less of an issue. It's still a concern, however, especially where managed code calls unmanaged APIs or COM objects.

2.3.5.1.1 Example of code injection through buffer overflows

An attacker can exploit a buffer overflow vulnerability to inject code. With this attack, a malicious user exploits an unchecked buffer in a process by supplying a carefully constructed input value that overwrites the program's stack and alters a function's return address. This causes execution to jump to the attacker's injected code.

The attacker's code usually ends up running under the process security context. This emphasizes the importance of using least privileged process accounts. If the current thread is impersonating, the attacker's code ends up running under the security context defined by the thread impersonation token. The first thing an attacker usually does is call the `RevertToSelf` API to revert to the process level security context that the attacker hopes has higher privileges.

2.3.5.1.2 Countermeasures to help prevent buffer overflows include:

- The first line of defense is to perform a thorough input validation. Although a bug may exist in your application that permits expected input to reach beyond the bounds of a container, unexpected input will be the primary cause of this vulnerability. Constrain input by validating it for type, length, format and range.
- When possible, limit your application's use of unmanaged code and thoroughly inspect the unmanaged APIs to ensure that input is properly validated.
- Inspect the managed code that calls the unmanaged API to ensure that only appropriate values can be passed as parameters to the unmanaged API.
- Use the `/GS` flag to compile Microsoft Visual C++ code. The `/GS` flag causes the compiler to inject security checks into the compiled code. This is not a fail proof solution or a replacement for your specific validation code; it does, however, protect your code from commonly known buffer overflow attacks.

2.3.5.2 Threat: Canonicalization

Different forms of input that resolve to the same standard name (the canonical name), is referred to as canonicalization. Code is particularly susceptible to canonicalization issues if it makes security decisions based on the name of a resource that is passed to the program as input. Files, paths and URLs are resource types that are vulnerable to canonicalization because in each case there are many different ways to represent the same name. File names are also problematic. For example, a single file could be represented as:

```
c:\temp\somefile.dat
somefile.dat
c:\temp\subdir\..\somefile.dat
..\somefile.dat
```

Ideally, your code does not accept input file names. If it does, the name should be converted to its canonical form prior to making security decisions, such as whether access should be granted or denied to the specified file.

Countermeasures to address canonicalization issues include:

- Avoid input file names where possible and instead use absolute file paths that cannot be changed by the end user.
- Make sure that file names are well formed (if you must accept file names as input) and validate them within the context of your application. For example, check that they are within your application's directory hierarchy.
- Make sure that the character encoding is set correctly to limit how input can be represented. Check that your application's Web.config has set the requestEncoding and responseEncoding attributes on the <globalization> element.

2.3.5.3 Threat: SQL injection

A SQL injection attack exploits vulnerabilities in input validation to run arbitrary commands in the database. It can occur when your application uses input to construct dynamic SQL statements to access the database. It can also occur if your code uses stored procedures that are passed strings that contain unfiltered user input. Using the SQL injection attack, the attacker can execute arbitrary commands in the database. The issue is magnified if the application uses an over privileged account to connect to the database. In this instance it's possible to use the database server to run operating system commands and potentially compromise other servers, in addition to being able to retrieve, manipulate and destroy data.

Your application may be susceptible to SQL injection attacks when you incorporate unvalidated user input into database queries. Particularly susceptible is code that constructs dynamic SQL statements with unfiltered user input. Consider the following code:

```
SqlDataAdapter myCommand = new SqlDataAdapter("SELECT * FROM Users WHERE  
UserName ='" + txtuid.Text + "'", conn);
```

Attackers can inject SQL by terminating the intended SQL statement with the single quote character followed by a semicolon character to begin a new command and then executing the command of their choice. Consider the following character string entered into the txtuid field.

```
'; DROP TABLE Customers –
```

This results in the following statement being submitted to the database for execution.

```
SELECT * FROM Users WHERE UserName=''; DROP TABLE Customers --'
```

This deletes the Customers table, assuming that the application's login has sufficient permissions in the database. This is another reason to use a least privileged login in the database. The double dash (--) denotes a SQL comment and is used to comment out any other characters added by the programmer, such as the trailing quote.

Other more subtle tricks can be performed. Supplying this input to the txtuid field:

```
' OR 1=1 –
```

This character string builds this command:

```
SELECT * FROM Users WHERE UserName=' ' OR 1=1 -
```

Because 1=1 is always true, the attacker retrieves every row of data from the Users table.

Countermeasures to prevent SQL injection include:

- Perform thorough input validation. Your application should validate its input prior to sending a request to the database.

- Use parameterized stored procedures for database access to ensure that input strings are not treated as executable statements. If you cannot use stored procedures, use SQL parameters when you build SQL commands.
- Use least privileged accounts to connect to the database.

2.3.5.4 Threat: XSS

An XSS attack can cause arbitrary code to run in a user's browser while the browser is connected to a trusted Web site. The attack targets your application's users and not the application itself, but it uses your application as the vehicle for the attack.

Because the script code is downloaded by the browser from a trusted site, the browser has no way of knowing that the code is not legitimate. IE security zones provide no defense. Since the attacker's code has access to the cookies associated with the trusted site, is the aim of the attack the attacked user's authentication cookie.

To initiate the attack, the attacker must convince the user to click on a carefully crafted hyperlink, for example, by embedding a link in an email sent to the user or by adding a malicious link to a forum posting. The link points to a vulnerable page in your application that echoes the unvalidated input back to the browser in the HTML output stream. For example, consider the following two links.

`www.webapp.com/logon.aspx?username=bob`

`www.webapp.com/logon.aspx?username=<script>alert('hacker code')</script>`

If the Web application takes the query string, fails to properly validate it and then returns it to the browser, the script code executes in the browser. The preceding example displays a harmless pop up message. With the appropriate script, the attacker can easily extract the user's authentication cookie, post it to his site and subsequently make a request to the target Web site as the authenticated user.

Countermeasures to prevent XSS include:

- Perform thorough input validation. Your applications must ensure that input from query strings, form fields and cookies are valid for the application. Consider all user input as possibly malicious and filter or sanitize for the context of the downstream code. Validate all input for known valid values and then reject all other input. Use regular expressions to validate input data received via HTML form fields, cookies and query strings.
- Use the `HTMLEncode` and `URLEncode` functions to encode any output that includes user input. This converts executable script into harmless HTML.

2.3.5.5 Design guidelines for preventing input validation attacks

Input validation is a challenging issue and the primary burden of a solution falls on application developers. However, proper input validation is one of your strongest measures of defense against today's application attacks. Proper input validation is an effective countermeasure that can help prevent XSS, SQL injection, buffer overflows and other input attacks.

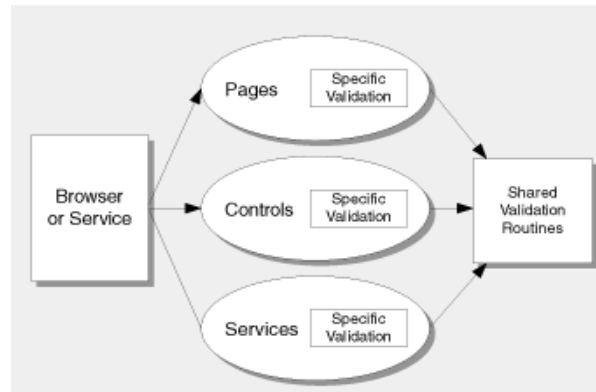
Input validation is challenging because there is not a single answer for what constitutes valid input across applications or even within applications. Likewise, there is no single definition of malicious input. Adding to this difficulty is that what your application does with this input influences the risk of exploit. For example, do you store data for use by other applications or does your application consume input from data sources created by other applications?

The following practices improve your Web application's input validation (Improving Web application security: Threats and Countermeasures):

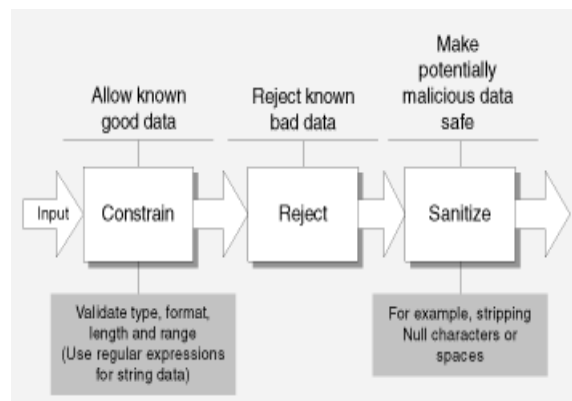
- **Assume all input is malicious.** Input validation starts with a fundamental supposition that all input is malicious until proven otherwise. Whether input comes from a service, a file share, a user or a database, validate your input if the source is outside your trust boundary. For example, if you call an external Web service that returns strings, how do you know that malicious commands are not present? Also, if several applications write to a shared database, when you read data, how do you know whether it's safe?

- **Centralize your approach.** Make your input validation strategy a core element of your application design. Consider a centralized approach to validation, for example, by using common validation and filtering code in shared libraries. This ensures that validation rules are applied consistently. It also reduces development effort and helps with future maintenance.

In many cases, individual fields require specific validation, for example, with specifically developed regular expressions. However, you can frequently factor out common routines to validate regularly used fields such as e-mail addresses, titles, names and postal addresses including ZIP or postal codes and so on. This approach is shown in figure, which has been taken from chapter 4 of *Improving Web application security: Threats and Countermeasures*, below.



- **Constrain, reject and sanitize your input.** The preferred approach to validating input is to constrain what you allow from the beginning. It's much easier to validate data for known valid types, patterns and ranges than it's to validate data by looking for known bad characters, since the range of valid data is generally a more finite set than potentially malicious data. However, for defense in depth you may also want to reject known bad input and then sanitize the input. The recommended strategy is shown in the figure, which has been taken from chapter 4 of *Improving Web application security: Threats and Countermeasures* below.



To create an effective input validation strategy, be aware of the following approaches and their tradeoffs:

- **Constrain input.** Constraining input is about allowing good data. This is the preferred approach. The idea here is to define a filter of acceptable input by using type, length, format and range. Define what is acceptable input, for your Web application's fields, and enforce it. Reject everything else as bad data.

Constraining input may involve setting character sets on the server so that you can establish the canonical form of the input in a localized way.

- **Validate data for type, length, format and range.** Use strong type checking on input data wherever possible. For example, use parameterized stored procedures for data access to benefit from strong type checking of input fields.

String fields should also be length and format checked. ZIP codes, personal identification numbers and so on have well defined formats that can be validated using regular expressions. Thorough checking is not only good programming practice; it makes it more difficult for an attacker to exploit your code. The attacker may get through your type check, but the length check may make executing his favorite attack more difficult.

- **Reject known bad input.** This approach is generally less effective than using the "allow" approach described earlier, but it's recommended to combine both. To deny bad data assumes your application knows all the variations of malicious input. Remember that there are multiple ways to represent characters. This is another reason why "allow" is the preferred approach.

While useful for applications that are already deployed and when you cannot afford to make significant changes, the "deny" approach is not as robust as the "allow" approach because bad data, such as patterns that can be used to identify common attacks, don't remain constant. Valid data remains constant while the range of bad data may change over time.

- **Sanitize input.** Sanitizing is about making potentially malicious data safe. It can be helpful when the range of input that is allowed cannot guarantee that the input is safe. This includes anything from stripping a null from the end of a user supplied string to escaping out values so they are treated as literals.

Another common example of sanitizing input in Web applications is using URL encoding or HTML encoding to wrap data and treat it as literal text rather than executable script. `HtmlEncode` methods escape out HTML characters and `UrlEncode` methods encode a URL so that it's a valid URI request.

- **Don't rely on client side validation.** Server side code should perform its own validation. What if an attacker bypasses your client, or shuts off your client side script routines, for example, by disabling JavaScript? Use client side validation to help reduce the number of round trips to the server but do not rely on it for security.
- **Be careful with canonicalization issues.** File paths and URLs are particularly prone to canonicalization issues and many well known exploits are a direct result of canonicalization bugs.

You should generally try to avoid designing applications that accept input file names from the user to avoid canonicalization issues. Consider alternative designs instead. For example, let the application determine the file name for the user.

If you do need to accept input file names, make sure they are strictly formed before making security decisions such as granting or denying access to the specified file.

2.3.5.6 How to implement these guidelines into practice

The following are examples applied to common input fields, using the preceding approaches:

- **Last name field.** This is a good example where constraining input is appropriate. In this case, you might allow string data in the range ASCII A – Z and a – z and also hyphens and curly apostrophes, observe that curly apostrophes have no significance to SQL, to handle names such as O'Dell. You would also limit the length to your longest expected value.
- **Quantity field.** This is another case where constraining input works well. In this example, you might use a simple type and range restriction. For example, the input data may need to be a positive integer between 0 and 1000.
- **Free text field.** Examples include comment fields on discussion boards. In this case, you might allow letters and spaces and also common characters such as apostrophes, commas and hyphens. The set that is allowed doesn't include less than and greater than signs, brackets and braces.

Some applications might allow users to mark up their text using a finite set of script characters, such as bold "``", italic "`<i>`", or even include a link to their favorite URL. In the case of a URL, your validation should encode the value so that it is treated as a URL.

- **An existing Web application that doesn't validate user input.** In an ideal scenario, the application checks for acceptable input for each field or entry point. However, if you have a Web application that doesn't validate user input, you need a stopgap approach to mitigate risk until you can improve your application's input validation strategy. While neither of the following approaches ensures safe handling of input, because that's dependent on where the input comes from and how it's used in your application, you should see them as quick fixes for a short term security improvement:
 - **HTML encoding and URL encoding user input when writing back to the client.** In this case, the assumption is that no input is treated as HTML and all output is written back in a protected form. This is sanitization in action.
 - **Rejecting malicious script characters.** This is a case of rejecting known bad input. In this case, a configurable set of malicious characters is used to reject the input. As described earlier, the problem with this approach is that bad data is a matter of context.

2.3.5.7 Input validation countermeasures in ASP .NET 1.X

Input validation is one of the most common and most devastating vulnerabilities in today's Web applications. Therefore it is extremely important not to make unfounded assumptions about the type, length, format or range of input.

2.3.5.7.1 Validation controls

In ASP .NET there is five input validation controls and one summary control. All the input validation controls inherits from the BaseValidator class, which in turn descends from the Web server control Label. They all have a series of properties and methods that are common to all validation controls. These properties and methods are:

- **ControlToValidate.** Gets or sets the input control to validate.
- **Display.** If client side validation is supported and enabled, this property gets or sets how the space error message should be allocated; none, statically or dynamically. In case of server side validation is this property ignored. A static display is possible only if the browser supports the display CSS style. The default is dynamic.
- **EnableClientScript.** True by default. Gets or sets whether client side validation is enabled.
- **Enabled.** Gets or sets whether the validation control is enabled.
- **ErrorMessage** - This is the message displayed in the validation summary.
- **ForeColor.** Gets or sets the color of the message displayed when validation fails.
- **IsValid** – Gets or sets a Boolean value for whether or not the associated input control passes validation or not.
- **Text.** Gets or sets the message shown when a validator fails. Note that this text doesn't replace the content of the ErrorMessage property in the validation summary.
- **Validate.** This method validates the associated input control and updates the IsValid property on both the page and on the individual validator.

All validator controls defined on a page are automatically grouped, on the server, in the validators collection of the Page class. You can validate all the controls at once using the Validate method of the Page class or individually by calling the Validate method on each validator.

Other than explicitly using the Validate method, the user's entry is also automatically validated whenever the page posts back. In this case, validation is performed when a button control is clicked. The button controls that can automatically apply validation are; Button, HtmlButton, HtmlInputImage, ImageButton and LinkButton. You can enable or disable a button's validation capability by using the CausesValidation property.

If the user is working with a browser that supports DHTML, the validation controls can perform client side validation using client script. This means that the page will not be posted back to the

server until all input fields contains valid data. If any of the validators are found to be in error, the submission of the form, to the server, is cancelled and the validator's Text property is displayed. This enhances the user experience by permitting the user to correct the input before submitting the form to the server. Field values are revalidated as soon as the field containing the error loses focus.

Note that the Web Forms page framework always performs validation on the server, even if the validation has already been performed on the client. This helps prevent users from being able to bypass validation by impersonating another user or a preapproved transaction. More information and an example of how client validation works can be found on page 160 in the book Programming Microsoft ASP.NET.

Observe that ASP.NET 1.1 passes the content of cookies, form fields and query strings through a regular expression, by default. This automatic behavior is controlled by the ValidateRequest attribute of the <pages> element in machine.config, which by default is set to true. To disable this behavior for all Web applications on your server, set the attribute to false. If you only want to enable this behavior for a specific Web application you can set the ValidateRequest attribute in the <pages> element of that Web application's Web.config file. If you want to fine grain it even further you can control this behavior on a per page basis using the new, in ASP .NET 1.1, @Page ValidateRequest attribute. Either way, the goal of this behavior is to raise a barrier against potentially dangerous script code injections. If a potentially dangerous client input value is detected, the processing of the request is aborted and an HTTP security exception is thrown. More specifically is the HttpRequestValidationException thrown. If your Web application needs to process script code it's possible to disable this behavior, but it's absolutely not recommended. You should also note that this shouldn't replace your own input validation.

Each validation control references an input control located somewhere on the page. When the page is going to be submitted, the contents of the monitored server control are passed to the validator for further processing. Each validator will perform different type of verification. Note that it's possible to use multiple validation controls with an individual input control.

Now it's time to look at the five input validation controls, found in ASP .NET 1.X.

2.3.5.7.1.1 The RequiredFieldValidation control

This control makes sure that the user doesn't skip an entry. A simple example could look like this:

```
Required field: <asp:textbox id="TextBox1" runat="server"/>
<asp:RequiredFieldValidator id="valRequired" runat="server"
    ControlToValidate=" TextBox1"
    ErrorMessage="* You must enter a value into TextBox1"
    Text="* You must enter a value into TextBox1">*
</asp:RequiredFieldValidator>
```

In the example above, there's a textbox which will not be valid until the user types something in it. If the user submits before entering anything to the textbox the message Required Field will show up and the message "You must enter a value into TB1" will be sent and possibly shown by the ValidationSummary control.

2.3.5.7.1.2 The CompareValidator control

This control allows you to compare the value entered by the user into an input control with the value entered into another input control or with a constant value.

This simple example shows how to make sure that the two textboxes are equal:

```
Textbox 1: <asp:textbox id=" TextBox1" runat="server"/>
Textbox 2: <asp:textbox id=" TextBox2" runat="server"/>
<asp:CompareValidator id="valCompare" runat="server"
    ControlToValidate=" TextBox1"
    ControlToCompare=" TextBox2"
    Operator=Equal
    ErrorMessage="* You must enter the same values into textbox 1 and 2"
    Text="* You must enter the same values into textbox 1 and 2">*
</asp:CompareValidator>
```

In this example we have two textboxes that must be equal. The tags that are unique to this control are the ControlToCompare attribute which is the control that will be compared. The two controls are compared with the type of comparison specified in the Operator attribute. The Operator attribute can contain Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual and DataTypeCheck.

Another usage of the CompareValidator is to compare a control to a value. The following code snippet shows how this is done.

```
Field: <asp:textbox id=" TextBox1" runat="server"/>
<asp:CompareValidator id="valRequired" runat="server"
  ControlToValidate=" TextBox1"
  ValueToCompare="50"
  Type="Integer"
  Operator="GreaterThan"
  ErrorMessage="* You must enter the a number greater than 50"
  Text="* You must enter the a number greater than 50"> *
</asp:CompareValidator>
```

The only new attribute in the example above is the data type (Type), which can be; Currency, Double, Date, Integer or String. The default data type is string.

2.3.5.7.1.3 The RangeValidator control

This control checks to see if a control value is within a valid range. The attributes that are necessary to this control are MaximumValue, MinimumValue and Type. In the following example are the Type value Date.

```
Enter a date between 1998 and 1999:
<asp:textbox id=" TextBox1" runat="server"/>
<asp:RangeValidator id="valRange" runat="server"
  ControlToValidate=" TextBox1"
  MaximumValue="1/1/1999"
  MinimumValue="1/1/1998"
  Type="Date"
  ErrorMessage="* The date must be between 1/1/1998 and 1/1/1999"
  Text="* The date must be between 1/1/1998 and 1/1/1999"> *
</asp:RangeValidator>
```

In this simple example is the date in TextBox1 validated to see if it's between 1/1/1998 – 1/1/1999.

2.3.5.7.1.4 The RegularExpressionValidator control

The RegularExpressionValidator control is used to determine whether the value of an input control matches a pattern defined by a regular expression. For example, using regular expression you can validate the format of e-mail addresses, telephone numbers, postal codes and so on. When using the RegularExpressionValidator control you set the ValidationExpression property with the regular expression, which will be used to validate the input. Here's an example of how a check of a Swedish "personnummer" could be done:

```
Personnummer: <asp:textbox id="TextBox1" " runat="server"/>
<asp:RegularExpressionValidator id="valRange" runat="server"
  ControlToValidate=" TextBox1"
  ValidationExpression="\d{6}-{4}"
  ErrorMessage="Invalid personnummer"
  Text="Invalid personnummer"> *
</asp:RegularExpressionValidator>
```

The example above validates the format of the supplied input field. First it type checks the input and then it checks the length. The input must consist of six numeric digits followed by a dash and then four digits. More information on regular expressions can be found at MSDN: Regular expressions.

If you are not using server controls, which rules out the use of validation controls, you can use the class `Regex`, found in the `System.Text.RegularExpressions` namespace. The following example shows how to validate the same field by using the static `Regex.IsMatch` method directly in the page class rather than using a validator control:

```
if(!Regex.IsMatch(TextBox1.Text, @"\d{6}-\d{4}"))
{
    //Code for invalid personummer
}
```

2.3.5.7.1.5 The *CustomValidator* control

This control is a generic and totally user defined validator that uses custom validation logic to accomplish its task. You typically resort to this control when none of the others validators seems appropriate.

To set up a custom validator , you set the `ClientValidationFunction` property to a client side function. If client side validation is disabled you just omit this setting. Alternatively or in addition to client validation, you can specify define some managed code to execute on the server. You do this by defining a handler for the `ServerValidate` event. This code will be executed when the page is posted back in response to a click on a button control. The following example shows how to configure a custom validator control to check the value of a text box against an array of possible values:

```
<asp:textbox id="TextBox1" runat="server"/>
<asp:CustomValidator runat="server" id="valCustom"
    ControlToValidate=" TextBox1"
    ClientValidationFunction="ClientValidation"
    OnServerValidate="ServerValidation"
    ErrorMessage="The Membership must be either Normal or Gold"
    Text=" The Membership must be either Normal or Gold"> *
</asp:CustomValidator>
```

Setting only client side validation code opens a security hole, because an attacker could work around the validation logic and manage to have invalid or malicious data sent to the server. By defining a server event handler you have one more chance to validate data before applying changes to the back end system. The server event handler might look something like this:

```
void ServerValidation(object source, ServerValidateEventArgs e)
{
    switch(e.Value)
    {
        case "Normal":    e.IsValid="true";
                        break;
        case "Gold":     e.IsValid="true";
                        break;
        default:         e.IsValid="false";
                        break;
    }
}
```

Observe that the `ServerValidateEventArgs` object contains two properties; `IsValid` and `Value`. The `Value` property is the value stored in the input control to be validated and the `IsValid` property must be set to either true or false according to the result of the validation.

2.3.5.7.1.6 The *ValidationSummary* control

The `ValidationSummary` control will collect all the error messages of all the non valid controls and display them after the page posts back. An example of this control could look like this:

```
<asp:ValidationSummary id="valSummary" runat="server"
    HeaderText="Errors:"
    ShowSummary="true"
    DisplayMode="List"/>
```

The error message list can either be shown on the web page, as shown in the example above, and / or in a popup box. To show the error messages in a popup box you specify the ShowMessageBox property to true. If you only want a popup box you remove the ShowSummary property.

2.3.5.7.2 Sanitizing input

Sanitizing is about making potentially malicious data safe. It can be helpful when the range of allowable input cannot guarantee that the input is safe. This might include stripping of a null from the end of a user supplied string or escaping values so they are treated as literals. If you need to sanitize input and covert or strip specific input characters, use Regex.Replace. An example of this method could be to strip out the following range of potentially unsafe characters; <, >, \, ", ', %, ;, (,) and &:

```
private string SanitizeInput(string input)
{
    Regex replaceBadCharsWithGood = new Regex(@"([<>""%';()&])");
    string goodChars = replaceBadCharsWithGood.Replace(input, "");
    return goodChars;
}
```

2.3.5.8 .NET countermeasures for XSS attacks

.NET offers two countermeasures to prevent XSS attacks; validate input and encode output.

How to validate input has just been described, so how do you encode the output.

If you write text output to a Web page and you don't know with absolute certainty that the text doesn't contain HTML special characters, such as <, > and &, then make sure to preprocess it using the HttpUtility.HtmlEncode method. It's recommended to do this even if the text came from user input, a database or a local file. The HttpUtility.HtmlEncode method replaces characters such as < with < and " with ". The encoded data doesn't cause the browser to execute code, instead is the data rendered as harmless HTML.

Similarly, use HttpUtility.UrlEncode method to encode URL strings. This method does like the HttpUtility.HtmlEncode method, replace special characters to ensure that all browsers will correctly transmit text in URL strings. Characters such as ?, &, / and spaces may be truncated or corrupted by some browsers so those characters cannot be used in ASP .NET pages in <A> tags or in query strings where the strings may be resent by a browser in a request string.

You should also know that no data bound Web controls encode their output except the TextBox control when it's TextMode property is set to MultiLine. If you bind any other control to data that has malicious XSS code, the code will be executed on the client. Therefore it's recommended to encode the data before you pass it back to the client.

If your Web page includes a free format text box in which you want to permit certain safe HTML elements such as and <i>, you can handle this safely by first preprocessing with the HtmlEncode method and then manually remove the encoding on the selected elements. An example of how to replace the and <i> elements are presented below:

```
StringBuilder sb = new StringBuilder(HttpUtility.HtmlEncode(userInput));
sb.Replace("&lt;b&gt;", "<b>")
sb.Replace("&lt;/b&gt;", "</b>")
sb.Replace("&lt;i&gt;", "<i>")
sb.Replace("&lt;/i&gt;", "</i>")
```

Another important countermeasure for XSS attacks is to set the correct character encoding. To successfully restrict what data is valid for your Web pages, it's important to limit the ways in which the input data can be represented. This prevents malicious users from using canonicalization and multibyte escape sequences to trick your input validation routines.

ASP .NET allows you to specify the character set at the page level or at the application level by using the <globalization> element in Web.config. Both approaches are shown below using the ISO-8859-1 character encoding, which is the default in early versions of HTML and HTTP.

To set the character encoding at the page level, use the <meta> element or the ResponseEncoding page level attribute as follows:

```
<meta http-equiv="Content Type" content="text/html; charset=ISO-8859-1" />
```

or

```
<% @ Page ResponseEncoding="ISO-8859-1" %>
```

To set the character encoding in Web.config, use the following configuration:

```
<configuration>
  <system.web>
    <globalization
      requestEncoding="ISO-8859-1"
      responseEncoding="ISO-8859-1"/>
  </system.web>
</configuration>
```

In addition to the above mentioned countermeasures, does VS .NET allow you to use some extra features of IE. These features are the HttpOnly cookie option and the <frame> and <iframe> element's security attribute.

IE 6.0 SP1 supports a new HttpOnly cookie attribute, which prevents client side scripts from accessing the cookie via the document.cookie property. Instead, an empty string is returned. The cookie is still sent to the server whenever the user browses to a Web site in the current domain.

Observe that Web browsers that don't support the HttpOnly cookie attribute either ignore the cookie or ignore the attribute, which means it's still a subject to XSS attacks.

The System.Net.Cookie class doesn't currently support an HttpOnly property. To add an HttpOnly attribute to the cookie, you need to add the following code to your Web application's Application_EndRequest event handler in Global.asax:

```
protected void Application_EndRequest(object sender, EventArgs e)
{
    string authcookie = FormsAuthentication.FormsCookieName;
    foreach(string sCookie in Response.Cookies)
    {
        //Just set the HttpOnly attribute on the Forms authentication cookie
        if(sCookie.Equals(authCookie))
        {
            //Force HttpOnly to be added to the cookie header
            Response.Cookies[sCookie].Path += ";HttpOnly";
        }
    }
}
```

IE 6.0 and later supports the second feature, which is the <frame> and <iframe> elements security attribute. You can use the security attribute to apply the user's Restricted Sites IE security zone settings to an individual <frame> or <iframe> element. By default, the Restricted Sites zone doesn't support script execution. If you use the security attribute, it must currently be set to "restricted" as shown below:

```
<frame security="restricted" src=http://www.somesite.com/somepage.htm></frame>
```

2.3.5.9 Input validation countermeasures in ASP .NET 2.0

ASP .NET 2.0 presents four new features to the input validation countermeasures.

The first new feature is the System.Net.Cookie class, which in Microsoft .NET Framework 2.0, supports an HttpOnly attribute. You can set this property programmatically or by using the <httpCookies> element in your application's Web.config file. By default, the HttpOnly cookie property isn't set, which can be seen in the machine.config.comments file's httpCookies element.

```
<httpCookies httpOnlyCookies="false" requireSSL="false" domain="" />
```

To set the `HttpOnly` property of the cookie, add the following entry in your application's `Web.config` file.

```
<system.web>
  <httpCookies httpOnlyCookies="true" requireSSL="false" domain=""/>
</system.web>
```

Earlier versions of the .NET Framework (versions 1.0 and 1.1) require that you added code to the `Application_EndRequest` event handler in your application `Global.asax` file, as shown above, to explicitly set the `HttpOnly` attribute.

Observe that you, with the `httpCookies` element, set the `Secure` and `HttpOnly` attributes on all cookies, including session cookies, within your `.config` file's folder. If you want to use different attributes to different cookies use a modified version of the code presented above to give different cookies different attributes.

The second new feature is the new `ValidationGroup` property of the form, button and validation controls in ASP .NET 2.0. You can assign a unique group name, through the `ValidationGroup` property, to one or more validation controls, which, for instance, are assigned to validate a `TextBox` control, and to a button control, which is the control that triggers the validation. This ensures that only the validation controls that belong to the same `ValidationGroup` as the clicked button will validate their associated controls. This enables you to have multiple control groups that are validated separately on the same page.

When the page is submitted the `Page.IsValid` property is set to true if there's no validation errors associated with the form controls in the validation group submitted. Observe that you should check the `Page.IsValid` property, as shown below, so that you don't run into problems when using validation with "Downlevel" browsers, such as Netscape.

```
void ButtonClick(Object s, EventArgs e)
{
    if(Page.IsValid)
    {
        Label1.Text = "The Page is valid";
    }
}
```

In ASP .NET 1.X, you can only call the `Page.Validate` method, which checks all validators controls in the page, without parameters. In ASP .NET 2.0 there is an overloaded `Validate` method, which takes a `ValidationGroup` property as its parameter. When you call this `Validate` method it validates against those controls whose `ValidationGroup` property equals the one passed to the `Validate` method. When a `Button` control causes a postback and its `CausesValidation` property is set to `True`, internally the `Page.Validate(ValidationGroupPropertyValue)` method gets called.

As in version 1.X, the `Page.Validators` property returns a collection of all validation controls. The overloaded `GetValidators(ValidationGroup)` method returns a collection of validation controls that belong to the specified validation group.

The third new feature in ASP .NET 2.0 is the new `ValidateEmptyText` property, which fixes an "issue" with the `CustomValidator` control. In previous versions of ASP .NET, if the targeted control's `Text` property had a value of `String.Empty`, the `CustomValidator`, together with all the other validators except the `RequiredFieldValidator`, would not evaluate the targeted control and would simply return that the validation passed. The news is that if the `ValidateEmptyText` property is set to true, the `CustomValidator` evaluates the control's value (using the criteria specified to the `CustomValidator` control) no matter what and returns the validation results. This property allows developers to evaluate the results of a `CustomValidator` control regardless of the value of the targeted control.

The fourth and final change has to do with the client validation. One of the unfortunate aspects of ASP .NET 1.X's validation controls was that they used a bit of proprietary JavaScript to provide client side validation functionality, making the validation controls inoperable in non Microsoft browsers. The workaround for version 1.x was to either Build up your own validation control library from scratch or use a free or third party validation package that has done the dirty work for you.

The client side script used by the validation controls in ASP .NET 2.0 still includes calls to `document.all`, but also includes equivalent checks using the standard compliant `document.getElementById(id)`. The end result is that the client side validation features of the validation controls now work with any browser that supports JavaScript 1.2 or higher.

2.3.6 Parameter manipulation

Parameter manipulation attacks are a class of attack that relies on the modification of the parameter data sent between the client and Web application. This includes query strings, form fields, cookies and HTTP headers. Top parameter manipulation threats include:

- Cookie manipulation
- Form field manipulation
- HTTP header manipulation
- Query string manipulation

2.3.6.1 Threat: Cookie manipulation

Cookies are susceptible to modification by the client. This is true for both persistent and memory resident cookies. A number of tools are available to help an attacker modify the contents of a memory resident cookie. Cookie manipulation is the attack that refers to the modification of a cookie, usually to gain unauthorized access to a Web site.

While SSL protects cookies over the network, it doesn't prevent them from being modified on the client computer. To counter the threat of cookie manipulation, encrypt or use an HMAC with the cookie.

2.3.6.2 Threat: Form field manipulation

The values of (X)HTML form fields are sent in plaintext to the server using the HTTP POST protocol. This may include visible and hidden form fields. Form fields of any type can be easily modified and client-side validation routines bypassed. As a result, applications that rely on form field input values to make security decisions on the server are vulnerable to attack.

To counter the threat of form field manipulation, instead of using hidden form fields, use session identifiers to reference state maintained in the state store on the server.

2.3.6.3 Threat: HTTP header manipulation

HTTP headers pass information between the client and the server. The client constructs request headers while the server constructs response headers. If your application relies on request headers to make a decision, your application is vulnerable to attack.

Don't base your security decisions on HTTP headers. For example, don't trust the HTTP Referer to determine where a client came from because this is easily falsified.

2.3.6.4 Threat: Query string manipulation

Users can easily manipulate the query string values passed by HTTP GET from client to server because they are displayed in the browser's URL address bar. If your application relies on query string values to make security decisions or if the values represent sensitive data the application is vulnerable to attacks.

Countermeasures to address the threat of query string manipulation include:

- Avoid using query string parameters that contain sensitive data or data that can influence the security logic on the server. Instead, use a session identifier to identify the client and store sensitive items in the session store on the server.
- Choose HTTP POST instead of GET to submit forms.
- Encrypt query string parameters.

2.3.6.5 Design guidelines for preventing parameter manipulation attacks

The following practices help secure your Web application's parameter manipulation (Improving Web application security: Threats and Countermeasures):

- **Encrypt sensitive cookie state.** Cookies may contain sensitive data such as session identifiers or data that is used as part of the server-side authorization process. To protect this type of data from unauthorized manipulation, use cryptography to encrypt the contents of the cookie.
- **Make sure that users don't bypass your checks.** Make sure that users don't bypass your checks by manipulating parameters. URL parameters can be manipulated by end users through the browser address text box. For example, the URL `http://www.<YourSite>/<YourApp>/sessionId=10` has a value of 10 that can be changed to some random number to receive different output. Make sure that you check this in server side code, not in client side JavaScript, which can be disabled in the browser.
- **Validate all values sent from the client.** Restrict the fields that the user can enter and modify and validate all values coming from the client. If you have predefined values in your form fields, users can change them and post them back to receive completely different results. Permit only known good values wherever possible. For example, if the input field is for a state, only inputs matching a state postal code should be permitted.
- **Don't trust HTTP header information.** HTTP headers are sent at the start of HTTP requests and HTTP responses. Your Web application should make sure it doesn't base any security decision on information in the HTTP headers because it's easy for an attacker to manipulate the header. For example, the referer field in the header contains the URL of the Web page from where the request originated. Don't make any security decisions based on the value of the referer field, for example, to check whether the request originated from a page generated by the Web application, because the field is easily falsified.

2.3.6.6 Parameter manipulation countermeasures in ASP .NET

Parameters, such as those found in form fields, query strings, view state and cookies, can be manipulated by attackers who usually intend to gain access to restricted pages or trick the application into performing an unauthorized operation.

The countermeasures that .NET offers are the following:

- Protect view state with MACs.
- Use `Page.ViewStateUserKey` to counter one click attacks.
- Maintain sensitive data on the server.
- Validate input parameters.

2.3.6.6.1 Protect view state with MACs

You can protect view state in two ways:

- First, you can set the `enableViewStateMac` attribute on the `<pages>` element in the `machine.config` file.
- Secondly, you can set the same attribute on the `@Page` directive. This allows you to customize settings on a per page basis.

If you set the `enableViewStateMac` attribute to `true` (the default value is `false`), you tell ASP .NET to run a message authentication code (MAC) check on the page's view state when the page is posted back from the client. This verifies that the view state hasn't been tampered with on the client.

2.3.6.6.2 Use Page.ViewStateUserKey to counter one click attacks

If you authenticate your callers and use view state, set the Page.ViewStateUserKey property in the Page_Init event handler to prevent one click attacks. A one click attack occurs when an attacker creates a prefilled Web page with view state. The attacker lures an unsuspecting user into browsing to the page, then causes the page to be sent to the server where the view state is valid. The server has no way of knowing that the view state originated from the attacker. View state validation and MACs don't counter this attack because the view state is valid and the page is executed under the security context of the user.

Set the Page.ViewStateUserKey property to a suitably unique value as a countermeasure to the one click attack. The value should be unique to each user and is typically a user name or identifier. When the attacker creates the view state, the ViewStateUserKey property is initialized to his or her name. When the user submits the page to the server, it's initialized with the attacker's name. As a result, the view state MAC check fails and an exception is generated.

2.3.6.6.3 Maintain sensitive data on the server

Don't trust input parameters, especially when they are used to make security decisions at the server. Also, don't use clear text parameters for any form of sensitive data. Instead, store sensitive data on the server in a session store and use a session token to reference the items in the store. Make sure that the user is authenticated securely and that the authentication token is secured properly.

2.3.6.6.4 Validate input parameters

Validate all input parameters that come from form fields, query strings, cookies and HTTP headers. More information about this can be found under the heading Input validation.

2.3.7 Sensitive data

Sensitive data is subject to a variety of threats. Attacks that attempt to view or modify sensitive data can target persistent data stores and networks. Top threats to sensitive data include:

- Access to sensitive data in storage
- Data tampering
- Network eavesdropping

2.3.7.1 Threat: Access to sensitive data in storage

You must secure sensitive data in storage to prevent a user, malicious or otherwise, from gaining access to and reading the data.

Countermeasures to protect sensitive data in storage include:

- Use restricted ACLs on the persistent data stores that contain sensitive data.
- Store encrypted data.
- Use identity and role based authorization to ensure that only the user or users with the appropriate level of authority are allowed access to sensitive data. Use role based security to differentiate between users who can view data and users who can modify data.

2.3.7.2 Threat: Data tampering

Data tampering refers to the unauthorized modification of data, often as it's passed over the network.

One countermeasure to prevent data tampering is to protect sensitive data passed across the network with tamper resistant protocols such as hashed message authentication codes (HMACs).

A HMAC provides message integrity in the following way:

1. The sender uses a shared secret key to create a hash based on the message payload.
2. The sender transmits the hash along with the message payload.
3. The receiver uses the shared key to recalculate the hash based on the received message payload. The receiver then compares the new hash value with the transmitted one. If they are the same, the message cannot have been tampered with.

2.3.7.3 Threat: Network eavesdropping

The HTTP data for Web application travels across networks in plaintext and is subject to network eavesdropping attacks, where an attacker uses network monitoring software to capture and potentially modify sensitive data.

Countermeasures to prevent network eavesdropping and to provide privacy include:

- Encrypt the data.
- Use an encrypted communication channel, for example, SSL.

2.3.7.4 Design guidelines for preventing Sensitive data attacks

Applications that deal with private user information such as credit card numbers, addresses, and medical records should take special steps to make sure that the data remains private and unaltered. In addition, secrets used by the application's implementation, such as passwords and database connection strings, must be secured. The security of sensitive data is an issue while the data is stored in persistent storage and while it is passed across the network. All guidelines can be found in chapter 4 of Improving Web application security: Threats and Countermeasures.

2.3.7.4.1 Secrets

Secrets include passwords, database connection strings and credit card numbers. The following practices improve the security of your Web application's handling of secrets:

- **Don't store secrets if you can avoid it.** Storing secrets in software in a completely secure fashion isn't possible. An administrator, who has physical access to the server, can access the data. For example, it's not necessary to store a secret when all you need to do is verify whether a user knows the secret. In this case, you can store a hash value that represents the secret and compute the hash using the user supplied value to verify whether the user knows the secret.
- **Don't store secrets in code.** Don't hard code secrets in code. Even if the source code is not exposed on the Web server, it's possible to extract string constants from compiled executable files. A configuration vulnerability may allow an attacker to retrieve the file.
- **Don't store database connections, passwords or keys in plaintext.** Avoid storing secrets such as database connection strings, passwords and keys in plaintext. Use encryption and store encrypted strings.
- **Avoid storing secrets in the Local Security Authority (LSA).** Avoid the LSA because your application requires administration privileges to access it. This violates the core security principle of running with least privilege. Also, the LSA can store secrets in only a restricted number of slots. A better approach is to use DPAPI, available on Microsoft Windows® 2000 and later operating systems.
- **Use Data Protection API (DPAPI) for encrypting secrets.** To store secrets such as database connection strings or service account credentials, use DPAPI. The main advantage to using DPAPI is that the platform system manages the encryption/decryption key and it's not an issue for the application. The key is either tied to a Windows user account or to a specific computer, depending on flags passed to the DPAPI functions. DPAPI is best suited for encrypting information that can be manually recreated when the master keys are lost, for example, because a damaged server requires an operating system reinstall. Data that cannot be recovered because you don't know the plaintext

value, for example, customer credit card details, require an alternate approach that uses traditional symmetric key based cryptography such as the use of 3DES.

2.3.7.4.2 Sensitive user data

Sensitive data such as logon credentials and application level data such as credit card numbers, bank account numbers and so on, must be protected. Privacy through encryption and integrity through message authentication codes (MAC) are the key elements.

The following practices improve your Web application's security of sensitive per user data:

- Retrieve sensitive data on demand.
- Encrypt the data or secure the communication channel and don't pass sensitive data using the HTTP GET protocol.
- Don't store sensitive data in persistent cookies.

2.3.7.4.2.1 Retrieve sensitive data on demand

The preferred approach is to retrieve sensitive data on demand when it's needed instead of persisting or caching it in memory. For example, retrieve the encrypted secret when it's needed, decrypt it, use it and then clear the memory (variable) used to hold the plaintext secret. If performance becomes an issue, consider the following options:

- **Cache the encrypted secret.** Retrieve the secret when the application loads and then cache the encrypted secret in memory, decrypting it when the application uses it. Clear the plaintext copy when it's no longer needed. This approach avoids accessing the data store on a per request basis.
- **Cache the plaintext Secret.** Avoid the overhead of decrypting the secret multiple times and store a plaintext copy of the secret in memory. This is the least secure approach but offers the optimum performance. Benchmark the other approaches before guessing that the additional performance gain is worth the added security risk.

2.3.7.4.2.2 Encrypt the data or secure the communication channel and don't pass sensitive data using the HTTP GET protocol

If you are sending sensitive data over the network to the client, encrypt the data or secure the channel. A common practice is to use SSL between the client and Web server. Between servers, an increasingly common approach is to use IPsec. For securing sensitive data that flows through several intermediaries, for example, Web service Simple Object Access Protocol (SOAP) messages, use message level encryption.

You should avoid storing sensitive data using the HTTP GET protocol because the protocol uses query strings to pass data. Sensitive data cannot be secured using query strings and query strings are often logged by the server.

2.3.7.4.2.3 Don't store sensitive data in persistent cookies

Avoid storing sensitive data in persistent cookies. If you store plaintext data, the end user is able to see and modify the data. If you encrypt the data, key management can be a problem. For example, if the key used to encrypt the data in the cookie has expired and been recycled, the new key cannot decrypt the persistent cookie passed by the browser from the client.

2.3.7.5 Sensitive data countermeasures in ASP .NET

For more information see the cryptography heading

2.3.8 Session management

Session management for Web applications is an application layer responsibility. Session security is critical to the overall security of the application.

Top session management threats include:

- Man in the middle
- Session hijacking
- Session replay

2.3.8.1 Threat: Man in the middle

A man in the middle attack occurs when the attacker intercepts messages sent between you and your intended recipient. The attacker then changes your message and sends it to the original recipient. The recipient receives the message, sees that it came from you and acts on it. When the recipient sends a message back to you, the attacker intercepts it, alters it and returns it to you. You and your recipient never know that you have been attacked.

Any network request involving client server communication, including Web requests, Distributed Component Object Model (DCOM) requests and calls to remote components and Web services, are subject to man in the middle attacks.

Countermeasures to prevent man in the middle attacks include:

- Use cryptography. If you encrypt the data before transmitting it, the attacker can still intercept it but cannot read it or alter it. If the attacker cannot read it, he or she cannot know which parts to alter. If the attacker blindly modifies your encrypted message, then the original recipient is unable to successfully decrypt it and, as a result, knows that it has been tampered with.
- Use Hashed Message Authentication Codes (HMACs). If an attacker alters the message, the recalculation of the HMAC at the recipient fails and the data can be rejected as invalid.

2.3.8.2 Threat: Session hijacking

A session hijacking attack occurs when an attacker uses network monitoring software to capture the authentication token (often a cookie) used to represent a user's session with an particular Web application. With the captured cookie, the attacker can spoof the user's session and gain access to the application. The attacker has the same level of privileges as the legitimate user.

Countermeasures to prevent session hijacking include:

- Use SSL to create a secure communication channel and only pass the authentication cookie over an HTTPS connection.
- Implement logout functionality to allow a user to end a session that forces authentication if another session is started.
- Make sure you limit the expiration period on the session cookie. Although this doesn't prevent session hijacking, it reduces the time window available to the attacker.

2.3.8.3 Threat: Session replay

Session replay occurs when a user's session token is intercepted and submitted by an attacker to bypass the authentication mechanism. For example, if the session token is in plaintext in a cookie or URL, an attacker can sniff it. The attacker then posts a request using the hijacked session token.

Countermeasures to help address the threat of session replay include:

- Reauthenticate when performing critical functions. For example, prior to performing a monetary transfer in a banking application, make the user supply the account password again.
- Expire sessions appropriately, including all cookies and session tokens.
- Create a "do not remember me" option to allow no session data to be stored on the client.

2.3.8.4 Design guidelines for preventing session management attacks

Web applications are built on the stateless HTTP protocol, so session management is an application level responsibility. Session security is critical to the overall security of an application.

The following practices improve the security of your Web application's session management (Improving Web application security: Threats and Countermeasures):

- Use SSL to protect session authentication cookies.
- Encrypt the contents of the authentication cookies.
- Limit session lifetime.
- Protect session state from unauthorized access.

2.3.8.4.1 Use SSL to protect session authentication cookies

Don't pass authentication cookies over HTTP connections. Set the secure cookie property within authentication cookies, which instructs browsers to send cookies back to the server only over HTTPS connections.

2.3.8.4.2 Encrypt the contents of the authentication cookies

Encrypt the cookie contents even if you are using SSL. This prevents an attacker viewing or modifying the cookie if he manages to steal it through an XSS attack. In this event, the attacker could still use the cookie to access your application, but only while the cookie remains valid.

2.3.8.4.3 Limit session lifetime

Reduce the lifetime of sessions to mitigate the risk of session hijacking and replay attacks. The shorter the session, the less time an attacker has to capture a session cookie and use it to access your application.

2.3.8.4.4 Protect session state from unauthorized access

Consider how session state is to be stored. For optimum performance, you can store session state in the Web application's process address space. However, this approach has limited scalability and implications in Web farm scenarios, where requests from the same user cannot be guaranteed to be handled by the same server. In this scenario, an out of process state store on a dedicated state server or a persistent state store in a shared database is required. ASP.NET supports all three options.

You should secure the network link from the Web application to state store using IPSec or SSL to mitigate the risk of eavesdropping. Also consider how the Web application is to be authenticated by the state store. Use Windows authentication where possible to avoid passing plaintext authentication credentials across the network and to benefit from secure Windows account policies.

2.3.8.5 Session Management countermeasures in ASP .NET

The two main factors, which you have to consider when providing secure session management, are the following:

- First, ensure that the session token cannot be used to gain access to sensitive pages where secure operations are performed or to gain access to sensitive data.
- Second, if the session data contains sensitive data, you must secure the session data, including the session store.

The following two types of tokens are associated with session management:

- **The session token.** This token is generated, by ASP .NET, automatically if session state is enabled. You enable session state by setting the mode attribute, of the <sessionState> element in machine.config, to InProc, SQLServer or StateServer. You can also override the <sessionState> configuration and disable or enable session state on a per page basis using the EnableSessionState attribute on the @Page tag.
- **The authentication token.** This is generated by authentication mechanisms, such as Forms authentication, to track an authenticated user's session. With a valid authentication token, a user can gain access to the restricted parts of your Web site.

The following countermeasures help you build secure session management:

- Require authentication for sensitive pages.
- Don't rely on client side state management options.
- Don't mix session tokens and authentication tokens.
- Use SSL effectively.
- Secure the session data.

2.3.8.5.1 Require authentication for sensitive pages

Make sure that you authenticate users before allowing them access to the sensitive and restricted parts of your site. If you use secure authentication and protect the authentication token with SSL, then a user's session is secure because an attacker cannot hijack and replay a session token.

How to secure the authentication token for Forms authentication is described earlier in the heading Authentication and Authorization.

2.3.8.5.2 Don't rely on client side state management options

Avoid using any of the client side state management options, such as view state, cookies, query strings or hidden form fields to store sensitive data. The information can be tampered with or seen in clear text. Use server side state management options, such as a database, to store sensitive data.

2.3.8.5.3 Don't mix session tokens and authentication tokens

Secure session management requires that you don't mix the two types of tokens. First, secure the authentication token to make sure an attacker cannot capture it and use it to gain access to the restricted areas of your application. Second, build your application in such a way that the session token alone cannot be used to gain access to sensitive pages or data. The session token should be used only for personalization purposes or to maintain the user state across multiple HTTP requests. Without authentication, don't maintain sensitive items of the user state.

2.3.8.5.4 Use SSL effectively

If your site has secure areas and public access areas, you must protect the secure authenticated areas with SSL. When a user moves back and forth between secure and public areas, the ASP .NET generated session cookie (or URL if you have enabled cookie less session state) moves with them in plaintext, but the authentication cookie is never passed over unencrypted HTTP connections as long as the Secure cookie property is set.

An attacker is able to obtain a session cookie passed over an unencrypted HTTP session, but if you have designed your site correctly and placed restricted pages in a separate and secure directory, the attacker can use it to access only to the non secure, public access pages. In this event, there is no security threat because these pages don't perform sensitive operations. Once the attacker tries to replay the session token to a secured page, the attacker is redirected to the application's login page because there is no authentication token.

2.3.8.5.5 Secure the session data

If the session data on the server contains sensitive items, the data and the store needs to be secured. ASP .NET supports several session state modes. How to secure these modes will not be discussed here, but you can found information about how to secure these modes in chapter 19 of the book *Improving Web application security: Threats and Countermeasures*.

3 Summary and conclusion

Since ASP .NET is such a huge topic cover I know that I have left out some parts that I really felt that should have been covered; such as assembly security. But due to time I had to limit myself, which is a little unfortunate. But the countermeasures that did make it into the thesis have been presented very thoroughly and should give the reader a very good start of building secure ASP .NET 1.X and 2.0 Web applications.

The thesis proves what I suspected from the beginning that the main goal of ASP .NET 2.0, which was to increase the productivity, has been achieved. The reason for this is that they have managed to simplify the development of ASP .NET Web applications quite dramatically with all the new prebuilt functionality.

The biggest security changes have been made to the countermeasures of the auditing and logging, authentication and authorization vulnerability categories. Although these changes haven't increased the security level of ASP .NET Web applications directly, they have simplified the work of securing ASP .NET Web applications enormously and therefore also increased the security level indirectly.

The most effective way of increasing productivity and security is to prebuild commonly used parts of a Web application. This is exactly what has been done in ASP .NET 2.0. As an example is the complete authentication feature now prebuilt and ready to be used. The only thing the developer needs to do is use the `DefaultOwinControl` on the Web applications login page, install the prebuilt database and make sure that the authentication feature is configured correctly. The drawback of this approach is of course that companies that already have a database with a specific structure need to rewrite parts of the authentication feature to make it work with their database. The positive thing is that it's relatively easy to do.

Other improvements made to ASP .NET 2.0 deal with some of the shortcomings in ASP .NET 1.X, such as the `HttpOnly` cookie attribute, AES encryption in the forms authentication, encrypting the `.config` files etc. The only major thing that really disappoints me is that there is no default authentication ticket / cookie logging to prevent cookie replay attacks.

The overall judgment of ASP .NET 2.0's security countermeasure must be positive although the improvements haven't dramatically improve the security level.

4 Future study and improvements

There are so much more to study when it comes to ASP .NET security, but due to time is this report limited to a small part of it.

The two things that would interest me the most to study further are the alternatives to forms authentication, such as Windows authentication and Microsoft's new web server IIS 7.0.

The thing that I really want a countermeasure for in a future release of .NET is cookie replay attacks.

5 Resources, Reference list

- Reference 1** **A First Look At ASP.NET v.2.0**
ISBN: 0321228960
Authors: Alex Homer, David Sussman, Rob Howard
Published by Addison – Wesley, 2003
- Reference 2** **Building Secure ASP.NET applications**
ISBN: 0- 7356 – 1890 – 9 (The book with .NET Framework 1.1)
Published by Microsoft Press, 2003 (The book with .NET Framework 1.1)
To download the book with .NET Framework 1.0, press [here](#)
To go to the Microsoft Press homepage for the book with the .NET Framework 1.1, press [here](#)
- Reference 3** **Defend your code with the top ten security tips every developer must know**
Authors: Keith Brown and Michael Howard
Published in the [September](#) issue of the [MSDN Magazine](#) 2002
To read the article, press [here](#)
- Reference 4** **Introducing Microsoft ASP.NET 2.0**
ISBN: 0 – 7356 – 2024 - 5
Author: Dino Esposito
Published by Microsoft Press, 2004
- Reference 5** **Improving Web application security: Threats and Countermeasures**
ISBN: 0 – 7356 – 1842 – 9
The book can be found [here](#)
- Reference 6** **Microsoft .NET Home**
To get there, press [here](#)
- Reference 7** **Microsoft Help and Support; Microsoft Knowledge Base article 313116**
To read this article, press [here](#)
- Reference 8** **Microsoft Operations Manager**
Home page can be found [here](#)
- Reference 9** **MSDN Library Website**
To get there press [here](#)

- Reference 10** **MSDN: ASP .NET 2.0 Internals**
The "article" can be found [here](#)
Observe that you must go to the heading ASP .NET 2.0 Internals yourself.
- Reference 11** **MSDN: How To: Instrument ASP .NET 2.0 Applications for Security**
To read it, press [here](#)
- Reference 12** **MSDN: Internet Information Services**
To read it, press [here](#)
- Reference 13** **MSDN: Regular expressions**
To read it, press [here](#)
- Reference 14** **Programming Microsoft ASP.NET**
ISBN: 0 – 7356 – 1903 – 4
Author: Dino Esposito
Publish by Microsoft Press, 2003
- Reference 15** **Understanding .NET: A Tutorial and Analysis**
ISBN: 0 – 201 – 74162 – 8
Author: David Chappell
Publish by Addison – Wesley, 2002
- Reference 16** **Anil John's blog about connection strings and Web farms**
To get to Anil John's blog, press [here](#)