Fortify Security Report

Jan 7, 2019

dlowe

Executive Summary

Issues Overview

On Jan 3, 2019, a source code review was performed over the Signal Performance Metrics code base. 1,824 files, 68,820 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 252 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
High	232
Critical	20

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:
Number of Files: 1824
Lines of Code: 68820
Build Label: <No Build Label>

Scan Information

Scan time: 01:21:19
SCA Engine version: 18.10.0187
Machine Name: SEDTS-L0G2-2VGJ
Username running scan: dlowe

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
DecodePeekLogs.Program.Main
DecodeSiemensLogs.Program.Main
DecodeTrafficwareLogs.Program.Main
FTPTimerService.Installer1.serviceInstaller1_AfterInstall
FTPTimerService.Installer1.serviceProcessInstaller1_AfterInstall
FTPfromAllControllers.FTPfromAllControllers.Main
FileByFileASC3Decoder.Program.Main
GenerateControllerEventLogObjectText.Program.Main
GetMaxTimeRecords.GetMaxTimeRecords.Main
ImportChecker.Program.Main
MOE.Common.Business.SignalFtp.OnSqlRowsCopied
MOE.Common.Data.LinkPivot.SchemaChanged
MOE.Common.Data.MOE.SchemaChanged
NEWDecodeandImportASC3Logs.Program.Main
NEWDecodeandImportASC3Logs.Properties.Settings.SettingChangingEventHandler
NEWDecodeandImportASC3Logs.Properties.Settings.SettingsSavingEventHandler
WavetronicsSpeedLibrary.TCPSpeedListener.ReceiveData
WavetronicsSpeedLibrary.UDPSpeedListener.ReceiveData

Private Information:
null.null.null

Java Properties:
DecodePeekLogs.Properties.Settings.get_BulkCopyBatchSize
DecodePeekLogs.Properties.Settings.get_BulkCopyTimeOut
DecodePeekLogs.Properties.Settings.get_Default
DecodePeekLogs.Properties.Settings.get_DeleteFiles
DecodePeekLogs.Properties.Settings.get_DestinationTableName
DecodePeekLogs.Properties.Settings.get_EarliestAcceptableDate
DecodePeekLogs.Properties.Settings.get_ForceNonParallel
DecodePeekLogs.Properties.Settings.get_MaxThreads
DecodePeekLogs.Properties.Settings.get_PeekCSVPAth
DecodePeekLogs.Properties.Settings.get_PeekDatPath
DecodePeekLogs.Properties.Settings.get_SPM
DecodePeekLogs.Properties.Settings.get_ToolsDirectory
DecodePeekLogs.Properties.Settings.get_WriteToConsole
DecodePeekLogs.Properties.Settings.get_decoderExePAth
DecodePeekLogs.Properties.Settings.get_gzipExePath
DecodeSiemensLogs.Properties.Settings.get_BulkCopyBatchSize
DecodeSiemensLogs.Properties.Settings.get_BulkCopyTimeOut
DecodeSiemensLogs.Properties.Settings.get_CSVOutPAth
DecodeSiemensLogs.Properties.Settings.get_DecoderPath
DecodeSiemensLogs.Properties.Settings.get_Default
DecodeSiemensLogs.Properties.Settings.get_DeleteFile
DecodeSiemensLogs.Properties.Settings.get_DeleteFiles
DecodeSiemensLogs.Properties.Settings.get_DestinationTableName
DecodeSiemensLogs.Properties.Settings.get_EarliestAcceptableDate
DecodeSiemensLogs.Properties.Settings.get_LogPath
DecodeSiemensLogs.Properties.Settings.get_MaxThreads
DecodeSiemensLogs.Properties.Settings.get_SPM
DecodeSiemensLogs.Properties.Settings.get_TimeOut
DecodeSiemensLogs.Properties.Settings.get_WriteToConsole
DecodeSiemensLogs.Properties.Settings.get_forceNonParallel
DecodeTrafficwareLogs.Properties.Settings.get_BulkCopyBatchSize
DecodeTrafficwareLogs.Properties.Settings.get_BulkCopyTimeOut
DecodeTrafficwareLogs.Properties.Settings.get_DecoderPath
DecodeTrafficwareLogs.Properties.Settings.get_Default
DecodeTrafficwareLogs.Properties.Settings.get_DeleteFile
DecodeTrafficwareLogs.Properties.Settings.get_DeleteFiles
DecodeTrafficwareLogs.Properties.Settings.get_DestinationTableName
DecodeTrafficwareLogs.Properties.Settings.get_EarliestAcceptableDate
DecodeTrafficwareLogs.Properties.Settings.get_MaxThreads
DecodeTrafficwareLogs.Properties.Settings.get_SPM
DecodeTrafficwareLogs.Properties.Settings.get_TWLogsPath
DecodeTrafficwareLogs.Properties.Settings.get_WriteToConsole
DecodeTrafficwareLogs.Properties.Settings.get_forceNonParallel
FTPTimerService.Properties.Settings.get_Default
FTPTimerService.Properties.Settings.get_WaitTimeInSeconds
GetMaxTimeRecords.Properties.Settings.get_BulkCopyBatchSize
GetMaxTimeRecords.Properties.Settings.get_BulkCopyTimeOut

GetMaxTimeRecords.Properties.Settings.get_Default
GetMaxTimeRecords.Properties.Settings.get_DestinationTableName
GetMaxTimeRecords.Properties.Settings.get_EarliestAcceptableDate
GetMaxTimeRecords.Properties.Settings.get_MaxThreads
GetMaxTimeRecords.Properties.Settings.get_SPM
GetMaxTimeRecords.Properties.Settings.get_WriteToConsole
GetMaxTimeRecords.Properties.Settings.get_forceNonParallel
MOE.Common.Properties.Settings.get_BulkCopyBatchSize
MOE.Common.Properties.Settings.get_BulkCopyTimeout
MOE.Common.Properties.Settings.get_Default
MOE.Common.Properties.Settings.get_SleepTime
MOE.Common.Properties.Settings.get_WriteToConsole
System.Configuration.ApplicationSettingsBase.get_Item
System.Configuration.ConfigurationManager.get_AppSettings

Stream:
AlexPilotti.FTPS.Common.FTPStream.Read
System.IO.BinaryReader.ReadChar
System.IO.FileStream.Read
System.IO.MemoryStream.Read
System.IO.MemoryStream.ReadByte
System.IO.Stream.Read
System.IO.StreamReader.ReadLine

System Information:
null.null.null
Lextm.SharpSnmpLib.SnmpException.ToString
System.Exception.ToString
System.Exception.get_Message
System.Configuration.ConfigurationManager.get_ConnectionStrings
System.Configuration.ConnectionStringSettings.ToString
System.Configuration.ConnectionStringSettings.get_ConnectionString
System.Configuration.ConnectionStringSettingsCollection.get_Item
System.Data.Common.DbConnection.get_DataSource
System.Data.Common.DbConnectionStringBuilder.get_ConnectionString
System.Data.SqlClient.SqlCommand.ExecuteReader
System.Net.Sockets.SocketException.get_Message
System.Runtime.InteropServices.ExternalException.ToString

Web:
System.Web.HttpRequestBase.get_UserHostAddress

Filter Set Summary

Current Enabled Filter Set:
Quick View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical

If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low
Visibility Filters:
If impact is not in range [2.5, 5.0] Then hide issue
If likelihood is not in range (1.0, 5.0] Then hide issue
If file is projects/github/atspm/spm/scripts/jquery-3.1.1.slim.js Then hide issue
If file is projects/github/atspm/spm/scripts/jquery-3.1.1.slim.min.js Then hide issue

Audit Guide Summary

Audit guide not enabled

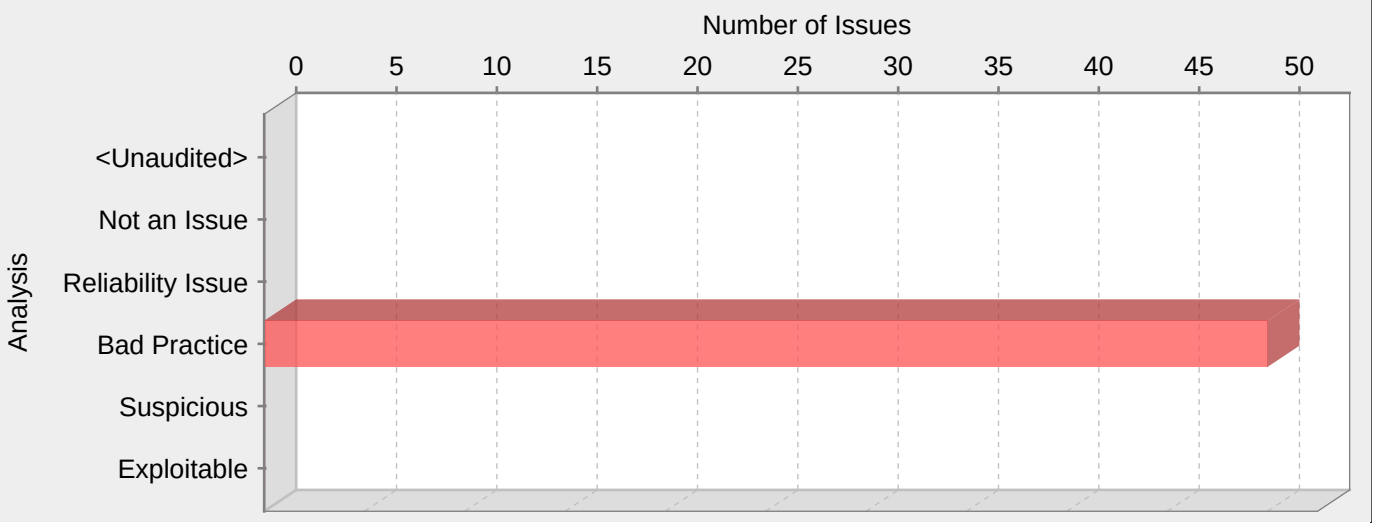
Results Outline

Overall number of results

The scan found 252 issues.

Vulnerability Examples by Category

Category: Mass Assignment: Insecure Binder Configuration (50 Issues)



Abstract:

The framework binder used for binding the HTTP request parameters to the model class has not been explicitly configured to allow, or disallow, certain attributes.

Explanation:

To ease development and increase productivity, most modern frameworks allow an object to be automatically instantiated and populated with the HTTP request parameters whose names match an attribute of the class to be bound. Automatic instantiation and population of objects speeds up development, but can lead to serious problems if implemented without caution. Any attribute in the bound classes, or nested classes, will be automatically bound to the HTTP request parameters. Therefore, malicious users will be able to assign a value to any attribute in bound or nested classes, even if they are not exposed to the client through web forms or API contracts.

Example 1: With no additional configuration, the following ASP.NET MVC controller method will bind the HTTP request parameters to any attribute in the RegisterModel or Details classes:

```
public ActionResult Register(RegisterModel model)
{
if (ModelState.IsValid)
{
try
{
return RedirectToAction("Index", "Home");
}
catch (MembershipCreateUserException e)
{
ModelState.AddModelError("", "");
}
}
return View(model);
}
```

Where RegisterModel class is defined as:

```
public class RegisterModel
{
[Required]
[Display(Name = "User name")]
```

```
public string UserName { get; set; }

[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[DataType(DataType.Password)]
[Display(Name = "Confirm password")]
public string ConfirmPassword { get; set; }

public Details Details { get; set; }

public RegisterModel()
{
Details = new Details();
}
}
```

and Details class is defined as:

```
public class Details
{
public Details()
{
IsAdmin = false;
}
public bool IsAdmin { get; set; }
...
}
```

Example 2: When using TryUpdateModel() or UpdateModel() in ASP.NET MVC or Web API applications, the model binder will automatically try to bind all HTTP request parameters by default:

```
public ViewResult Register()
{
var model = new RegisterModel();
TryUpdateModel<RegisterModel>(model);
return View("detail", model);
}
```

Example 3: In ASP.NET Web API applications, the model binder will automatically try to bind all HTTP request parameters by default using the configured JSON or XML serializer/deserializer. By default, the binder will try to bind all possible attributes from the HTTP request parameters or body:

```
public class ProductsController : ApiController
{
public string SaveProduct([FromBody] Product p)
{
return p.Name;
}
...
}
```

Example 4: In ASP.NET Web Form applications, the model binder will automatically try to bind all HTTP request parameters when using TryUpdateModel() or UpdateModel() with IValueProvider interface.

```
Employee emp = new Employee();
TryUpdateModel(emp, new System.Web.ModelBinding.FormValueProvider(ModelBindingExecutionContext));
if (ModelState.IsValid)
{
db.SaveChanges();
```

```
}
```

and Employee class is defined as:

```
public class Employee
{
public Employee()
{
IsAdmin = false;
IsManager = false;
}
public string Name { get; set; }
public string Email { get; set; }
public bool IsManager { get; set; }
public bool IsAdmin { get; set; }
}
```

Recommendations:

When using frameworks that provide automatic model binding capabilities, it is a best practice to control which attributes will be bound to the model object so that even if attackers are able to identify other non-exposed attributes of the model or nested classes, they will not be able to bind arbitrary values from HTTP request parameters.

Depending on the framework used there will be different ways to control the model binding process:

Example 5: It is possible to control the ASP.NET MVC model binding process using the [Bind] attribute:

```
[Bind(Exclude = "IsAdmin")]
public class Details
{
public Details()
{
IsAdmin = false;
}
public bool IsAdmin { get; set; }
...
}
```

Example 6: When using TryUpdateModel() or UpdateModel() in ASP.NET MVC or Web API applications, an additional string array can be passed to specify the allowed or disallowed attributes:

```
public ViewResult Register()
{
var model = new RegisterModel();
String[] array = new String[] { "FirstName" };
TryUpdateModel<RegisterModel>(model, array);
return View("detail", model);
}
```

Example 7: In ASP.NET Web API applications, the model classes bound to the HTTP request parameters can be annotated to control the binding process. For example, if the application is using JSON.NET for JSON serialization, the [JsonIgnore] attribute can be used to ignore certain attributes that should not be included in the serialization/deserialization process:

```
public class Details
{
public Details()
{
IsAdmin = false;
}
[JsonIgnore]
public bool IsAdmin { get; set; }
...
```

```
}
```

Example 8: In ASP.NET Web Form applications, this issue can be prevented using appropriate use of [BindNever] attribute, available under System.Web.ModelBinding namespace.

```
public class Employee
{
public Employee()
{
IsAdmin = false;
IsManager = false;
}
public string Name { get; set; }
public string Email { get; set; }
[BindNever]
public bool IsManager { get; set; }
[BindNever]
public bool IsAdmin { get; set; }
}
```

A different approach to protecting against mass assignment vulnerabilities is using a layered architecture where the HTTP request parameters are bound to DTO objects. The DTO objects are only used for that purpose, exposing only those attributes defined in the web forms or API contracts, and then mapping these DTO objects to Domain objects where the rest of the private attributes can be defined.

Tips:

1. This vulnerability category can be classified as a design flaw since accurately finding these issues requires understanding of the application architecture which is beyond the capabilities of static analysis. Therefore, it is possible that if the application is designed to use specific DTO objects for HTTP request binding, there will not be any need to configure the binder to exclude any attributes.

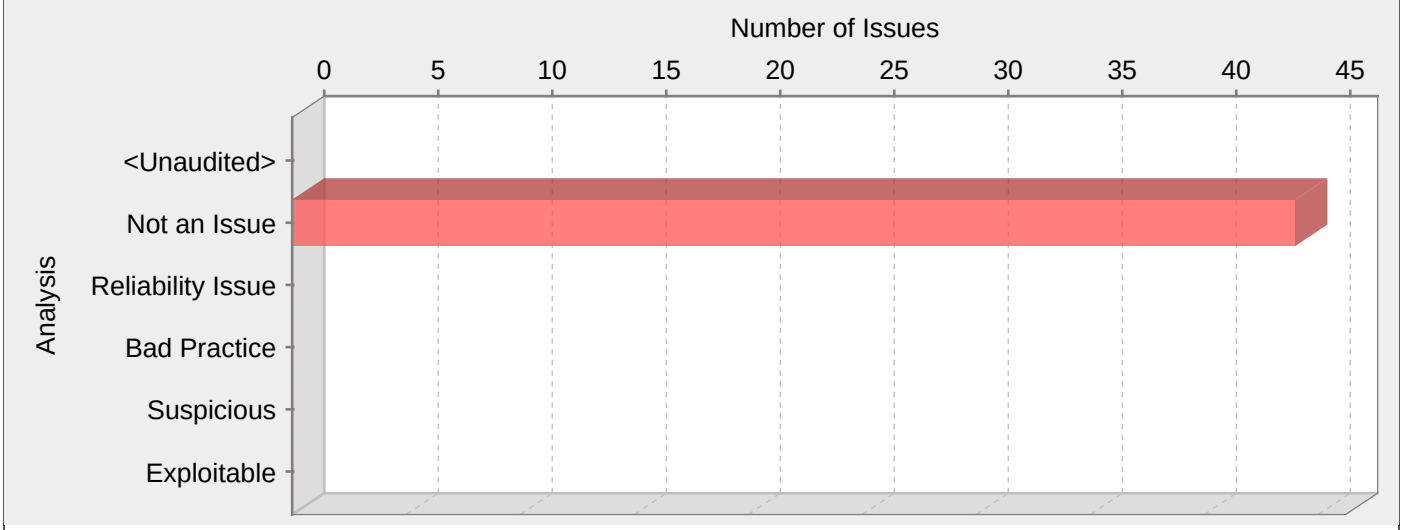
AccountController.cs, line 292 (Mass Assignment: Insecure Binder Configuration)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The framework binder used for binding the HTTP request parameters to the model class has not been explicitly configured to allow, or disallow, certain attributes.		
Sink:	AccountController.cs:292 Variable: model()		

```
290         [AllowAnonymous]
291         [ValidateAntiForgeryToken]
292         public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
293         {
294             if (returnUrl == null)
```

Analysis:	Bad Practice	
Comments:	*dlowe@utah.gov 2018-08-01 9:33 AM*	This code is from Microsoft. Need to investigate what properties to bind to.

Category: Password Management: Password in Configuration File (44 Issues)



Abstract:

Storing a plaintext password in a configuration file could result in a system compromise.

Explanation:

Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.

Recommendations:

A password should never be stored in plaintext. Instead, the password should be entered by an administrator when the system starts. If that approach is impractical, a less secure but often adequate solution is to obfuscate the password and scatter the de-obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

Microsoft(R) provides a tool that can be used in conjunction with the Windows Data Protection application programming interface (DPAPI) to protect sensitive application entries in configuration files [1].

Tips:

1. HPE Security Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password and that the password entry contains plaintext.

2. If the entry in the configuration file is a default password, require that it be changed in addition to requiring that it be obfuscated in the configuration file.

3. The following ASP .NET configuration file appears to contain an encrypted password with the hexadecimal value 0x174f7a68:

```
<configuration>
...
<appSettings>
<add key="pwd" value="0x174f7a68" />
</appSettings>
...
</configuration>
```

However, to verify that the password is actually encrypted, you must locate where the password is used in the application code and ensure that an appropriate decryption algorithm is used. If such a function is never called, then the creator of the password may have chosen a hexadecimal value to make the password hard to guess, which means the application is still vulnerable because the password is stored in plaintext.

app.config, line 9 (Password Management: Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Storing a plaintext password in a configuration file could result in a system compromise.		
Sink:	app.config:9 null()		

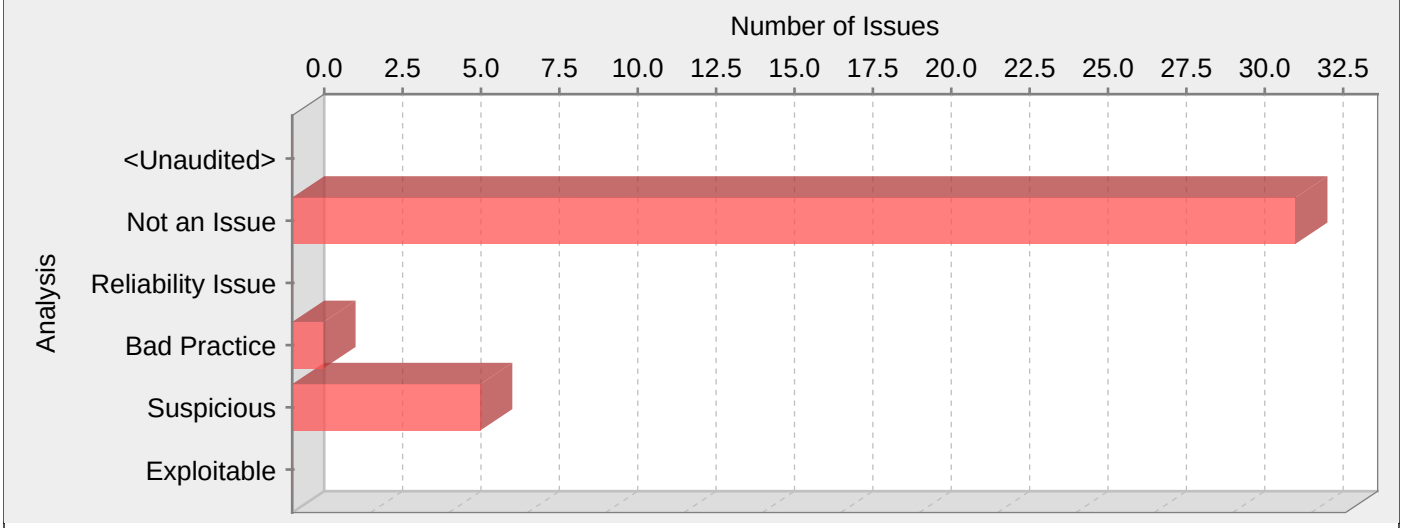
```
7    <connectionStrings>
```

```
8                       <!--<add name="SPM" connectionString="Data Source=MOE;Initial
                    Catalog=MOETest;Persist Security Info=True;integrated
                    security=True;MultipleActiveResultSets=True;Connection
                    Timeout=500;App=EntityFramework" providerName="System.Data.SqlClient" />-->
9                       <add name="SPM" connectionString="data source=AtspmServer;initial
                    catalog=MOE1;Persist Security Info=True;User ID=SPM;Password=******
                    providerName="System.Data.SqlClient" />
10                      <add name="ADConnectionString" connectionString="LDAP://itwcap1" />
11                    </connectionStrings>
```

Analysis: Not an Issue

Comments: *dlowe@utah.gov 2018-08-01 9:03 AM* These will be encrypted on the production machines

Category: ASP.NET MVC Bad Practices: Optional Submodel With Required Property (39 Issues)



Abstract:

The model class Approach has a required property of which the type is an optional member of a parent model type and therefore may be susceptible to under-posting attacks.

Explanation:

If a model class has required property and is the type of an optional member of a parent model class, it may be susceptible to under-posting attacks if an attacker communicates a request that contains less data than is expected.

The ASP.NET MVC framework will try to bind request parameters to model properties, including submodels.

If a submodel is optional -- that is, the parent model has a property without the [Required] attribute -- and if an attacker does not communicate that submodel, then the parent property will have a null value and the required fields of the child model will not be asserted by model validation. This is one form of an under-posting attack.

Consider the following the model class definitions:

```
public class ChildModel
{
public ChildModel()
{
}
[Required]
public String RequiredProperty { get; set; }
}
public class ParentModel
{
public ParentModel()
{
}
public ChildModel Child { get; set; }
}
```

If an attacker does not communicate a value for the ParentModel.Child property, then the ChildModel.RequiredProperty property will have a [Required] which is not asserted. This may produce unexpected and undesirable results.

Recommendations:

Make the child model field required:

```
public class ChildModel
{
public ChildModel()
{
}
[Required]
```

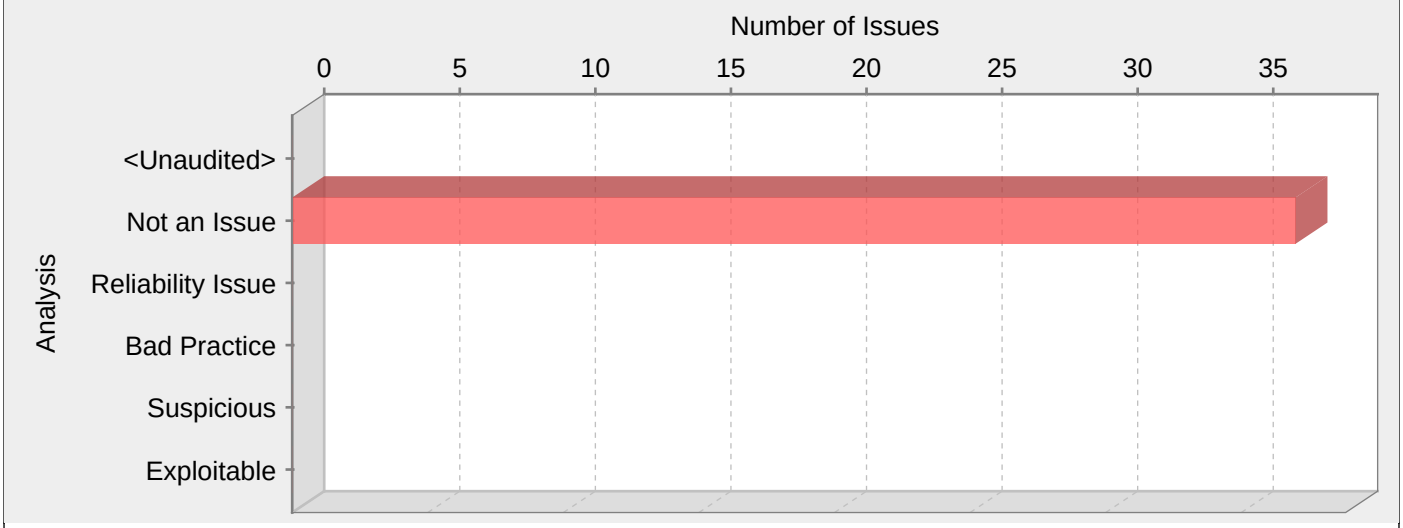
```
public String RequiredProperty { get; set; }
}
public class ParentModel
{
public ParentModel()
{
}
[Required]
public ChildModel Child { get; set; }
}
```

Approach.cs, line 26 (ASP.NET MVC Bad Practices: Optional Submodel With Required Property)			
Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The model class Approach has a required property of which the type is an optional member of a parent model type and therefore may be susceptible to under-posting attacks.		
Sink:	Approach.cs:26 Function: set_ApproachModel()		

```
24          public ControllerEventLogs DetectorEvents { get; private set; }
25
26          public Models.Approach ApproachModel { get; set; }
27
28          public void SetDetectorEvents(Models.Approach approach, DateTime startdate,
DateTime enddate, bool has_pcd,
```

Analysis:	Not an Issue		
Comments:	*dlowe@utah.gov 2018-07-31 1:45 PM*	These relationships are required for either partial views or to define a relationship in entity framework	

Category: ASP.NET MVC Bad Practices: Controller Action Not Restricted to POST (37 Issues)



Abstract:

The controller action in class ApproachesController may benefit from being restricted to only accept the POST verb.

Explanation:

ASP.NET MVC controller actions that modify data by writing, updating, or deleting could benefit from being restricted to accept the POST verb. This increases the difficulty of cross-site request forgery because accidental clicking of links will not cause the action to execute.

The following controller action by default accepts any verb and may be susceptible to cross-site request forgery:

```
public ActionResult UpdateWidget(Model model)
{
// ... controller logic
}
```

Recommendations:

The following controller action accepts only the POST verb because it has the [HttpPost] attribute:

```
[HttpPost]
public ActionResult UpdateWidget(Model model)
{
// ... controller logic
}
```

This approach does not completely protect against cross-site request forgery; sophisticated attackers can still use scripting-based approaches to circumvent this mechanism. However, using this approach increases the difficulty of an attack with very little cost.

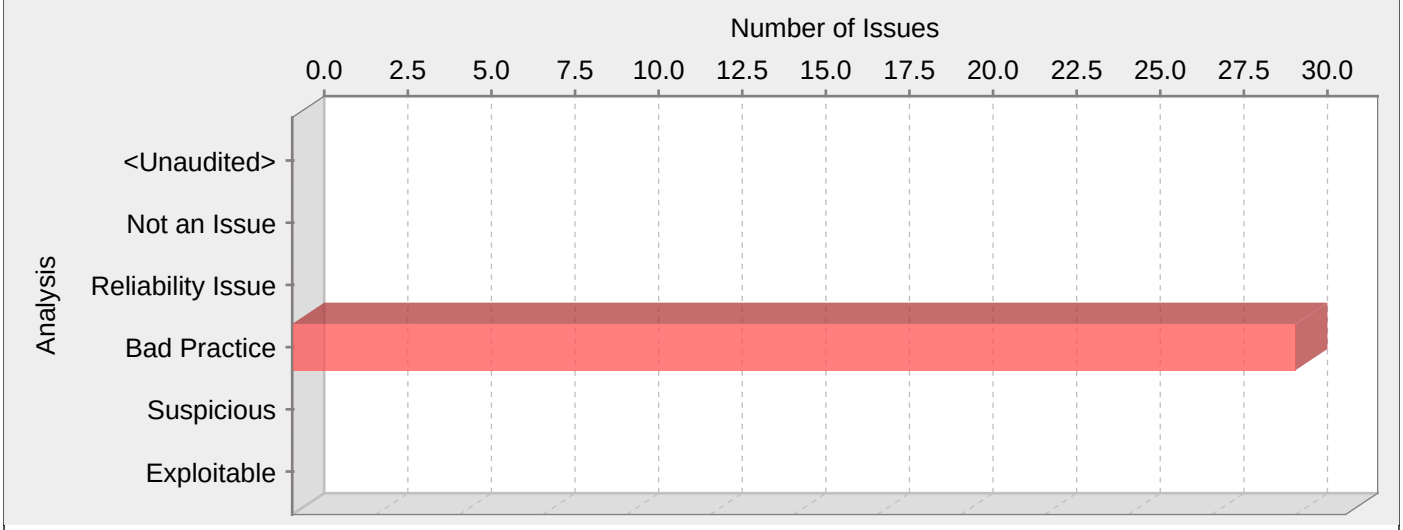
ApproachesController.cs, line 73 (ASP.NET MVC Bad Practices: Controller Action Not Restricted to POST)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The controller action in class ApproachesController may benefit from being restricted to only accept the POST verb.		
Sink:	ApproachesController.cs:73 Function: Delete()		

```
71          // GET: RouteSignals/Delete/5
72           [Authorize(Roles = "Admin, Configuration")]
73          public ActionResult Delete(int? id)
74          {
75              if (id == null)
```

Analysis:	Not an Issue		
Comments:	*dlowe 2019-01-04 1:10 PM*	This does not update any resources. Not a post method.	

Category: ASP.NET MVC Bad Practices: Model With Optional and Required Properties (30 Issues)



Abstract:

The model class AggregateDataExportController has properties that are required and properties that are not required and therefore may be susceptible to over-posting attacks.

Explanation:

Using a model class that has properties that are required (as marked with the [Required] attribute) and properties that are optional (as not marked with the [Required] attribute) can lead to problems if an attacker communicates a request that contains more data than is expected.

The ASP.NET MVC framework will try to bind request parameters to model properties.

Having mixed requiredness without explicitly communicating which parameters are to be model-bound may indicate that there are model properties for internal use but can be controlled by attacker.

The following code defines a possible model class that has properties that have [Required] and properties that do not have [Required]:

```
public class MyModel
{
[Required]
public String UserName { get; set; }
[Required]
public String Password { get; set; }
public Boolean IsAdmin { get; set; }
}
```

If any optional parameters can change the behavior of an application, then an attacker may be able to actually change that behavior by communicating an optional parameter in a request.

Recommendations:

There are a few possible mechanisms to prevent optional binding:

1. Define model classes specific to external communication.

The following code defines a model class where all properties have [Required], and without any optional properties:

```
public class MyRequiredModel
{
[Required]
public String UserName { get; set; }
[Required]
public String Password { get; set; }
}
```

2. Use the [Bind] attribute to control which parameters should be bound to the model object from the request.

The following code defines a possible model class that has properties that have [Required] and properties that do not have [Required] and has the [Bind] attribute to specify which parameters should be bound:

[Bind(Include="UserName,Password")]
public class MyModel
{
[Required]
public String UserName { get; set; }

[Required]
public String Password { get; set; }

public Boolean IsAdmin { get; set; }
}

The following code applies the [Bind] attribute at the controller action method:

[HttpPost]
public ActionResult Login([Bind(Include = "UserName,Password")] MyModel model)
{
// ... controller logic
}

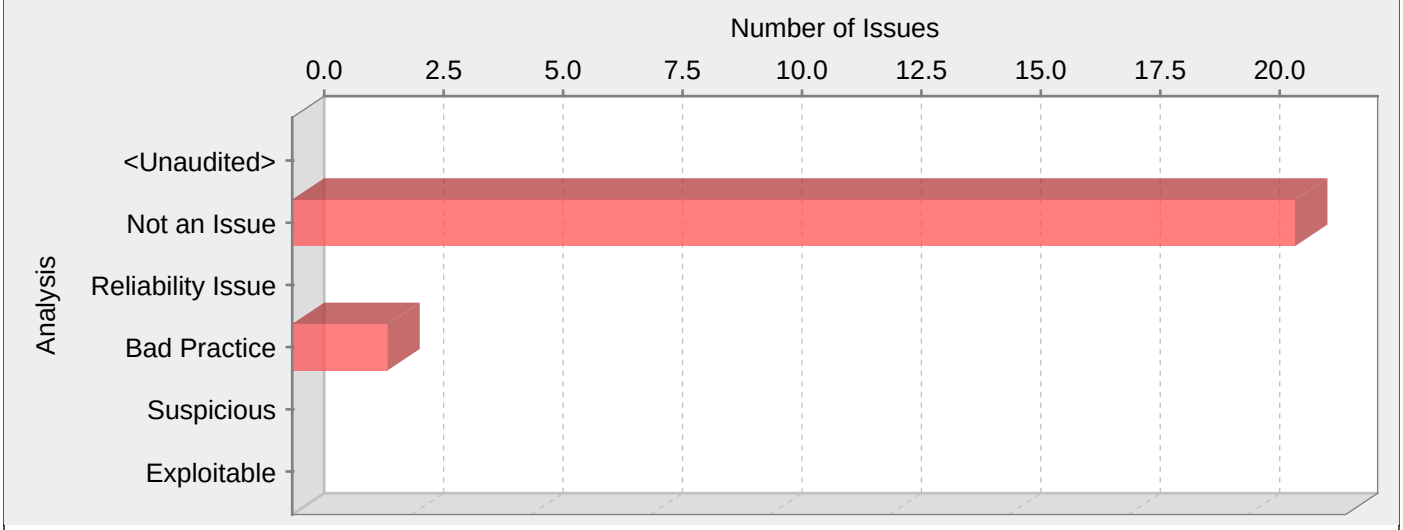
AggregateDataExportController.cs, line 79 (ASP.NET MVC Bad Practices: Model With Optional and Required Properties)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The model class AggregateDataExportController has properties that are required and properties that are not required and therefore may be susceptible to over-posting attacks.		
Sink:	AggregateDataExportController.cs:79 Function: CreateMetric()		

```
77          [HttpPost]
78          [ValidateAntiForgeryToken]
79          public ActionResult CreateMetric(AggDataExportViewModel
aggDataExportViewModel)
80          {
81              switch (aggDataExportViewModel.SelectedMetricTypeId)
```

Analysis:	Bad Practice

Category: ASP.NET MVC Bad Practices: Model With Required Non-Nullable Property (23 Issues)



Abstract:

The model class ApplicationEvent has a required non-nullable property and therefore may be susceptible to under-posting attacks.

Explanation:

Using a model class that has non-nullable properties that are required (as marked with the [Required] attribute) can lead to problems if an attacker communicates a request that contains less data than is expected.

The ASP.NET MVC framework will try to bind request parameters to model properties.

If a model has a required non-nullable parameter and an attacker does not communicate that required parameter in a request -- that is, the attacker uses an under-posting attack -- then the property will have the default value (usually zero) which will satisfy the [Required] validation attribute. This may produce unexpected application behavior.

The following code defines a possible model class that has a required enum, which is non-nullable:

```
public enum ArgumentOptions
{
OptionA = 1,
OptionB = 2
}
public class Model
{
[Required]
public String Argument { get; set; }
[Required]
public ArgumentOptions Rounding { get; set; }
}
```

Recommendations:

There are a few possible ways to address this problem:

1. Wrap non-nullable types in a Nullable. If an attacker does not communicate a value, then the property will be null and will not satisfy the [Required] validation attribute.

The following code defines a possible model class that wraps an enum with a Nullable (as with the ? after the type of the property):

```
public enum ArgumentOptions
{
OptionA = 1,
OptionB = 2
}
public class Model
{
[Required]
```

public String Argument { get; set; }

[Required]
public ArgumentOptions? Rounding { get; set; }
}

2. Define the default value of zero as a safe default or as a known invalid value.

The following code defines a possible model class that has a required enum, which is non-nullable, but has a safe default value:

public enum ArgumentOptions
{
Default = 0,
OptionA = 1,
OptionB = 2
}

public class Model
{
[Required]
public String Argument { get; set; }

[Required]
public ArgumentOptions Rounding { get; set; }
}

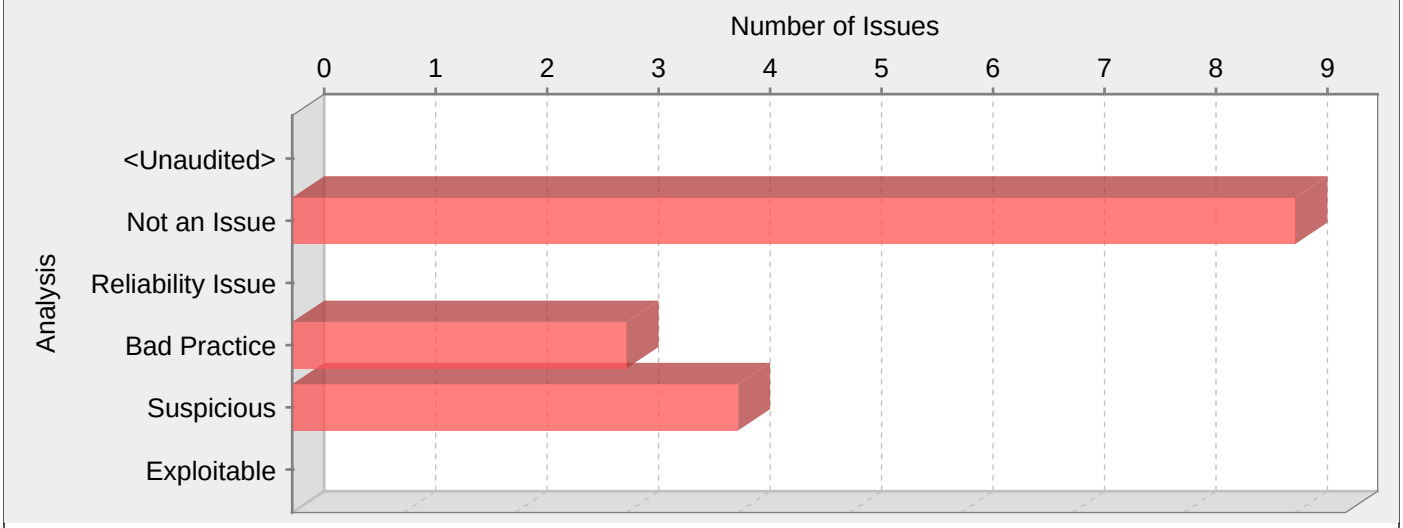
ApplicationEvent.cs, line 20 (ASP.NET MVC Bad Practices: Model With Required Non-Nullable Property)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The model class ApplicationEvent has a required non-nullable property and therefore may be susceptible to under-posting attacks.		
Sink:	ApplicationEvent.cs:20 Function: set_Timestamp()		

```
18
19          [Required]
20          public DateTime Timestamp { get; set; }
21
22          [Required]
```

Analysis:	Not an Issue		
Comments:	*dlowe@utah.gov 2018-08-01 10:41 AM*	We do not want nullable dates in the database	

Category: Path Manipulation (16 Issues)



Abstract:

Attackers can control the filesystem path argument to FileStream() at FTPSClient.cs line 737, which allows them to access or modify otherwise protected files.

Explanation:

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the filesystem.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker may provide a file name like "..\\..\\Windows\\System32\\krnl386.exe", which will cause the application to delete an important Windows system file.

```
String rName = Request.Item("reportName");
...
File.delete("C:\\users\\reports\\" + rName);
```

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension ".txt".

```
sr = new StreamReader(resmngr.GetString("sub")+".txt");
while ((line = sr.ReadLine()) != null) {
Console.WriteLine(line);
}
```

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the HPE Security Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Implementation of an effective blacklist is notoriously difficult. One should be skeptical if validation logic requires blacklisting. Consider different types of input encoding and different sets of metacharacters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. ASP.NET Request Validation and WCF are among them. To highlight the unvalidated sources of input, the HPE Security Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HPE Security Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. In case of ASP.NET Request Validation, we also provide evidence for when validation is explicitly disabled. We refer to this feature as Context-Sensitive Ranking. To further assist the HPE Security Fortify user with the auditing process, the HPE Security Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

FTPSClient.cs, line 737 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the filesystem path argument to FileStream() at FTPSClient.cs line 737, which allows them to access or modify otherwise protected files.		

Source: FTPSClient.cs:1530 System.Net.Sockets.TcpClient.GetStream()

```
1528                }
1529                else
1530                    s = dataClient.GetStream();
1531
1532                return s;
```

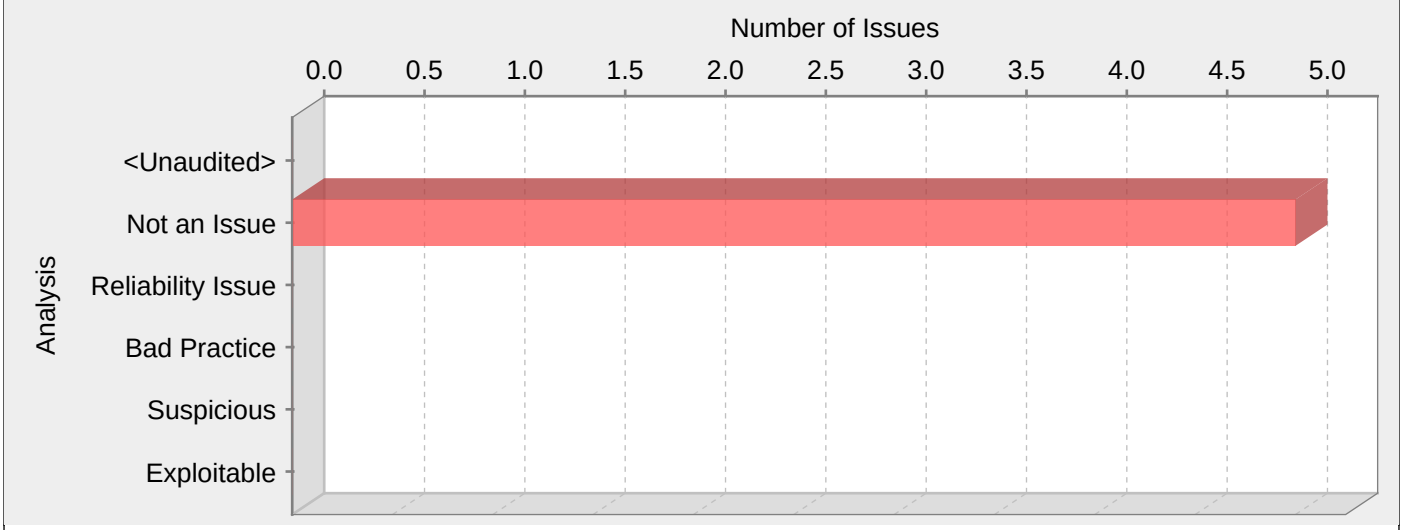
Sink: FTPSClient.cs:737 System.IO.FileStream.FileStream()

```
735                 using (Stream s = GetFile(remoteFileName))
736                 {
737                     using (FileStream fs = new FileStream(localFileName, FileMode.Create, FileAccess.Write, FileShare.None))
738                     {
739                         byte[] buf = new byte[1024];
```

Analysis: Bad Practice

Comments:	*dlowe@utah.gov 2018-07-31 12:48 PM*	This code was downloaded through Microsoft NUGET. The program that uses ftps is a scheduled task command line tool. This runs behind a firewall and is not accessible to the outside world. Also the program does not use this specific function.

Category: Password Management: Hardcoded Password (5 Issues)



Abstract:

Hardcoded passwords may compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following code uses a hardcoded password to create a network credential:

```
...
NetworkCredential netCred =
new NetworkCredential("scott", "tiger", domain);
...
```

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is likely no way to change the network credential user "scott" with a password of "tiger" unless the program is patched. An employee with access to this information could use it to break into the system. Even worse, if attackers have access to the executable for the application they can disassemble the code, which will contain the values of the passwords used.

Recommendations:

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

Microsoft(R) provides a tool that can be used in conjunction with the Windows Data Protection application programming interface (DPAPI) to protect sensitive application entries in configuration files [1].

Tips:

1. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the HPE Security Fortify Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

FTPSClient.cs, line 503 (Password Management: Hardcoded Password)

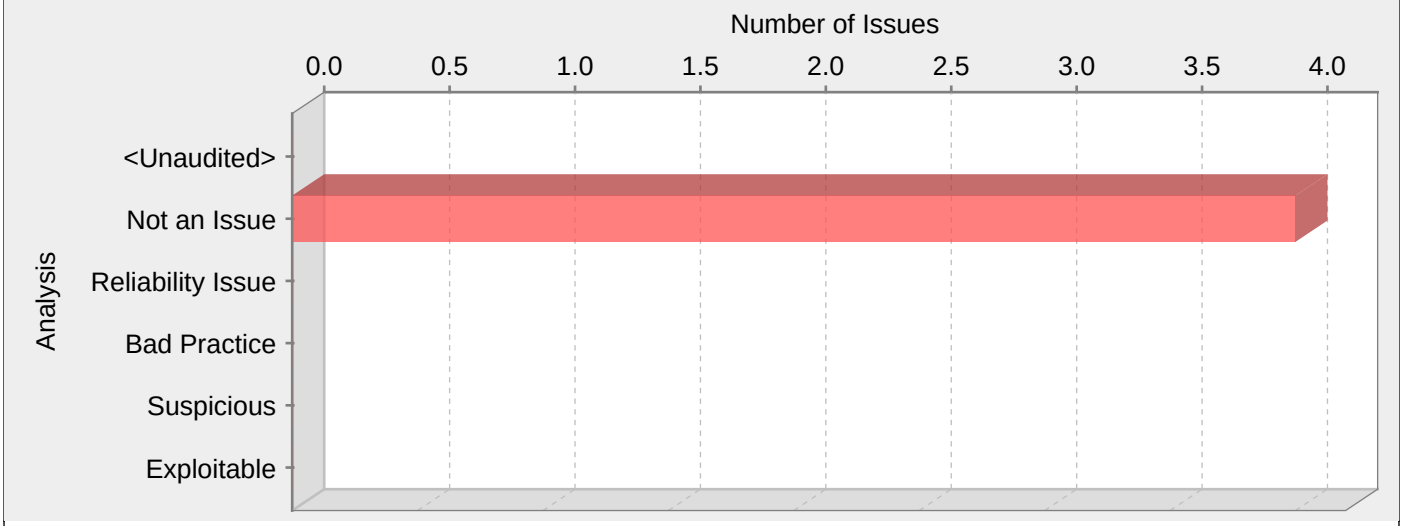
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords may compromise system security in a way that cannot be easily remedied.		

Sink: FTPSClient.cs:503 NetworkCredential()

```
501                 // Anonymous authentication
502                 if (credential == null)
503                     credential = new NetworkCredential(anonUsername, anonPassword);
504
505                 if (timeout != null)
```

Analysis: Not an Issue

Category: Open Redirect (4 Issues)



Abstract:

The file AccountController.cs passes unvalidated data to an HTTP redirect on line 697. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Explanation:

Redirects allow web applications to direct users to different pages within the same application or to external sites. Applications utilize redirects to aid in site navigation and, in some cases, to track how users exit the site. Open redirect vulnerabilities occur when a web application redirects clients to any arbitrary URL that can be controlled by an attacker.

Attackers can utilize open redirects to trick users into visiting a URL to a trusted site and redirecting them to a malicious site. By encoding the URL, an attacker can make it more difficult for end-users to notice the malicious destination of the redirect, even when it is passed as a URL parameter to the trusted site. Open redirects are often abused as part of phishing scams to harvest sensitive end-user data.

Example 1: The following code instructs the user's browser to open a URL parsed from the dest request parameter when a user clicks the link.

```
String redirect = Request["dest"];
Response.Redirect(redirect);
```

If a victim receives an email instructing the user to follow a link to "http://trusted.example.com/ecommerce/redirect.asp?dest=www.wilyhacker.com", the user might click on the link believing they would be transferred to the trusted site. However, when the user clicks the link, the code above will redirect the browser to "http://www.wilyhacker.com".

Many users have been educated to always inspect URLs they receive in emails to make sure the link specifies a trusted site they know. However, if the attacker encoded the destination url as follows:

"http://trusted.example.com/ecommerce/redirect.asp?dest=%77%69%6C%79%68%61%63%6B%65%72%2E%63%6F%6D"

then even a savvy end-user may be fooled into following the link.

Recommendations:

Unvalidated user input should not be allowed to control the destination URL in a redirect. Instead, use a level of indirection: create a list of legitimate URLs that users are allowed to specify and only allow users to select from the list. With this approach, input provided by users is never used directly to specify a URL for redirects.

Example 2: The following code references an array populated with valid URLs. The link the user clicks passes in the array index that corresponds to the desired URL.

```
String redirect = Request["dest"];
Int32 strDest = System.Convert.ToInt32(redirect);
if((strDest >= 0) && (strDest <= strURLArray.Length -1 ))
{
strFinalURL = strURLArray[strDest];
pageContext.forward(strFinalURL);
}
```

In some situations this approach is impractical because the set of legitimate URLs is too large or too hard to keep track of. In such cases, use a similar approach to restrict the domains that users can be redirected to, which can at least prevent attackers from sending users to malicious external sites.

Tips:

1. A number of modern web frameworks provide mechanisms for performing validation of user input. ASP.NET Request Validation and WCF are among them. To highlight the unvalidated sources of input, the HPE Security Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HPE Security Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. In case of ASP.NET Request Validation, we also provide evidence for when validation is explicitly disabled. We refer to this feature as Context-Sensitive Ranking. To further assist the HPE Security Fortify user with the auditing process, the HPE Security Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

2. Fortify RTA adds protection against this category.

AccountController.cs, line 697 (Open Redirect)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The file AccountController.cs passes unvalidated data to an HTTP redirect on line 697. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.		

Source: AccountController.cs:292 Login(1)

```
290         [AllowAnonymous]
291         [ValidateAntiForgeryToken]
292         public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
293         {
294             if (returnUrl == null)
```

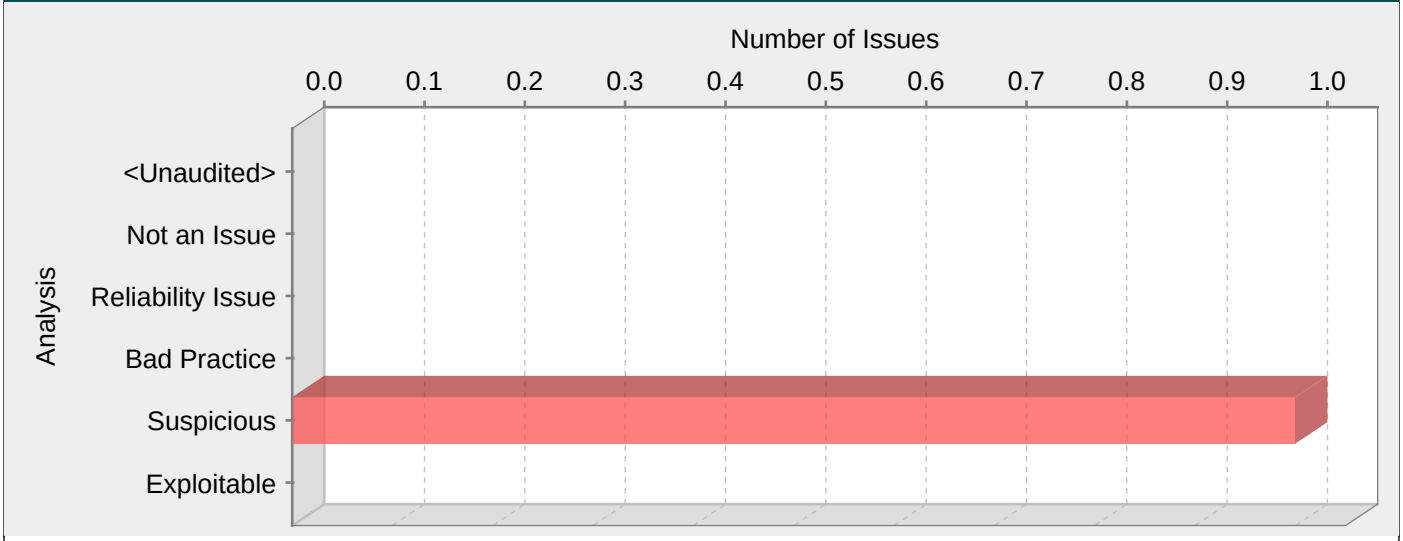
Sink: AccountController.cs:697 System.Web.Mvc.Controller.Redirect()

```
695             if (Url.IsLocalUrl(returnUrl))
696             {
697                 return Redirect(returnUrl);
698             }
699             return RedirectToAction("Index", "Home");
```

Analysis: Not an Issue

Comments:	*dlowe@utah.gov 2018-07-31 12:32 PM*	I have changed the code to verify that the return url is a local url.

Category: Dangerous Function: xp_cmdshell (1 Issues)



Abstract:

The function xp_cmdshell cannot be used safely. It should not be used.

Explanation:

Certain functions behave in dangerous ways regardless of how they are used. The function xp_cmdshell launches a Windows command shell to execute the provided command string. The command executes either in the default system or a provided proxy context. However, there is no way to limit a user to prespecified set of privileged operations and any privilege grant opens up the user to execute any command string.

Recommendations:

Any privileged operations that need to be performed should be exposed in a way that limits access. Possible alternatives are external assemblies that perform only the necessary actions or stored procedures that make use of impersonation contexts.

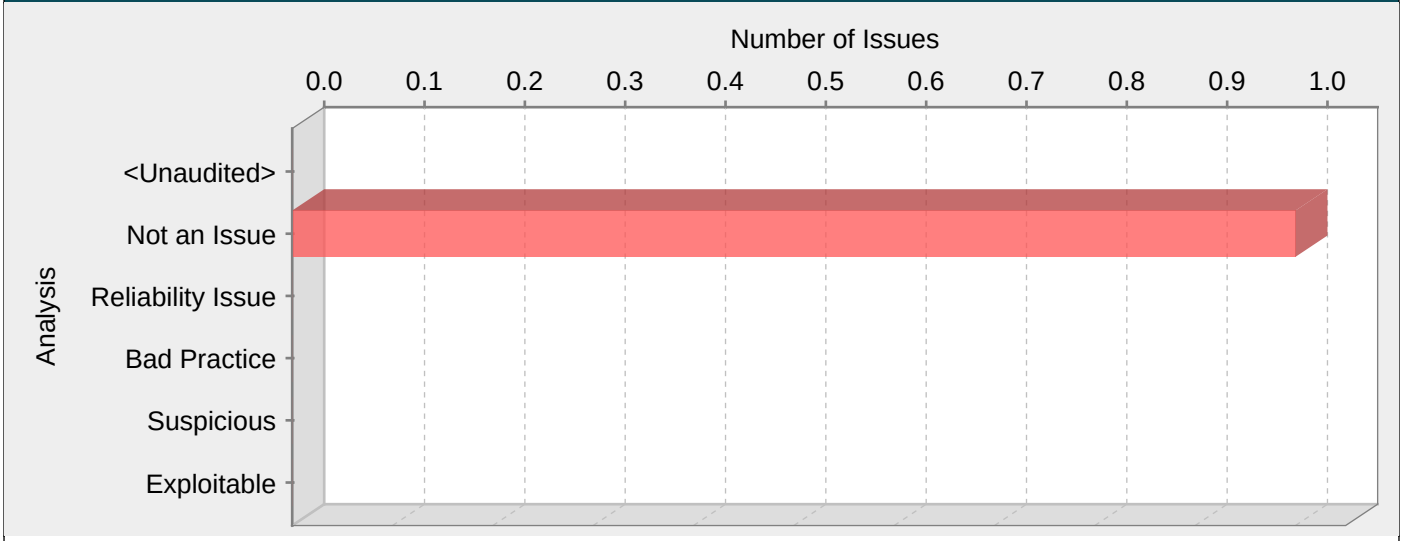
PreserveData.sql, line 77 (Dangerous Function: xp_cmdshell)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	API Abuse		
Abstract:	The function xp_cmdshell cannot be used safely. It should not be used.		
Sink:	PreserveData.sql:77 xp_cmdshell()		

```
75                 @Notes = 'Before bcp Command, plan about ten hours.'
76               END
77               EXEC master..xp_cmdshell @SQLSTATEMENT
78
79               SET @SQLSTATEMENT = 'DROP TABLE [dbo].[' + @StagingTableName  + ']'
```

Analysis:	Suspicious		
Comments:	*dlowe 2019-01-04 11:24 AM*	Currently this is required during the install process. This statement has to run through a dos command to work. On the plus side this will only be executed during installs and upgrades	

Category: JSON Injection (1 Issues)



Abstract:

On line 111 of DataExportController.cs, the method GetReCaptchaStatus() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.

Explanation:

JSON injection occurs when:

1. Data enters a program from an untrusted source.

2. The data is written to a JSON stream.

Applications typically use JSON to store data or send messages. When used to store data, JSON is often treated like cached data and may potentially contain sensitive information. When used to send messages, JSON is often used in conjunction with a RESTful service and can be used to transmit sensitive information such as authentication credentials.

The semantics of JSON documents and messages can be altered if an application constructs JSON from unvalidated input. In a relatively benign case, an attacker may be able to insert extraneous elements that cause an application to throw an exception while parsing a JSON document or request. In a more serious case, such as that involving JSON injection, an attacker may be able to insert extraneous elements that allow for the predictable manipulation of business critical values within a JSON document or request. In some cases, JSON injection can lead to cross-site scripting or dynamic code evaluation.

Example 1: The following C# code uses JSON.NET to serialize user account authentication information for non-privileged users (those with a role of "default" as opposed to privileged users with a role of "admin") from user-controlled input variables username and password to the JSON file located at C:\user_info.json:

```
...
StringBuilder sb = new StringBuilder();
StringWriter sw = new StringWriter(sb);
using (JsonWriter writer = new JsonTextWriter(sw))
{
writer.Formatting = Formatting.Indented;
writer.WriteStartObject();
writer.WritePropertyName("role");
writer.WriteRawValue("\"default\"");
writer.WritePropertyName("username");
writer.WriteRawValue("\"" + username + "\"");
writer.WritePropertyName("password");
writer.WriteRawValue("\"" + password + "\"");
writer.WriteEndObject();
}
File.WriteAllText(@"C:\user_info.json", sb.ToString());
```

Yet, because the JSON serialization is performed using JsonWriter.WriteRawValue(), the untrusted data in username and password will not be validated to escape JSON-related special characters. This allows a user to arbitrarily insert JSON keys, possibly changing the structure of the serialized JSON. In this example, if the non-privileged user mallory with password Evil123! were to append ","role":"admin to her username when entering it at the prompt that sets the value of the username variable, the resulting JSON saved to C:\user_info.json would be:

{
"role":"default",
"username":"mallory",
"role":"admin",
"password":"Evil123!"
}

If this serialized JSON file were then deserialized to a Dictionary object with JsonConvert.DeserializeObject() as so:

String jsonString = File.ReadAllText(@"C:\user_info.json");

Dictionary<string, string> userInfo = JsonConvert.DeserializeObject<Dictionary<string, strin>>(jsonString);

The resulting values for the username, password, and role keys in the Dictionary object would be mallory, Evil123!, and admin respectively. Without further verification that the deserialized JSON values are valid, the application will incorrectly assign user mallory "admin" privileges.

Recommendations:

When writing user supplied data to JSON some guidelines should be followed:

1. Don't create JSON attributes whose names are derived from user input.

2. Ensure that all serialization to JSON is performed using a safe serialization function that delimits untrusted data within single or double quotes and escapes any special characters.

Example 2: The following C# code implements the same functionality as that in Example 1, but instead uses JsonWriter.WriteValue() rather than JsonWriter.writeRawValue() to serialize the data, therefore ensuring that any untrusted data is properly delimited and escaped:

...

StringBuilder sb = new StringBuilder();
StringWriter sw = new StringWriter(sb);

using (JsonWriter writer = new JsonTextWriter(sw))
{
writer.Formatting = Formatting.Indented;

writer.WriteStartObject();

writer.WritePropertyName("role");
writer.WriteValue("default");

writer.WritePropertyName("username");
writer.WriteValue(username);

writer.WritePropertyName("password");
writer.WriteValue(password);

writer.WriteEndObject();
}

File.WriteAllText(@"C:\user_info.json", sb.ToString());

DataExportController.cs, line 111 (JSON Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 111 of DataExportController.cs, the method GetReCaptchaStatus() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.		
Source:	DataExportController.cs:109 System.Net.WebClient.DownloadString()		

```
107                 string secretKey = settings.ReCaptchaSecretKey;
108                 var client = new WebClient();
```

```
109                         var result = client.DownloadString(
110                             $"https://www.google.com/recaptcha/api/siteverify?secret={secretKey}&response={respons
                            e}&remoteip={userIp}");
111                         var obj = JObject.Parse(result);
```

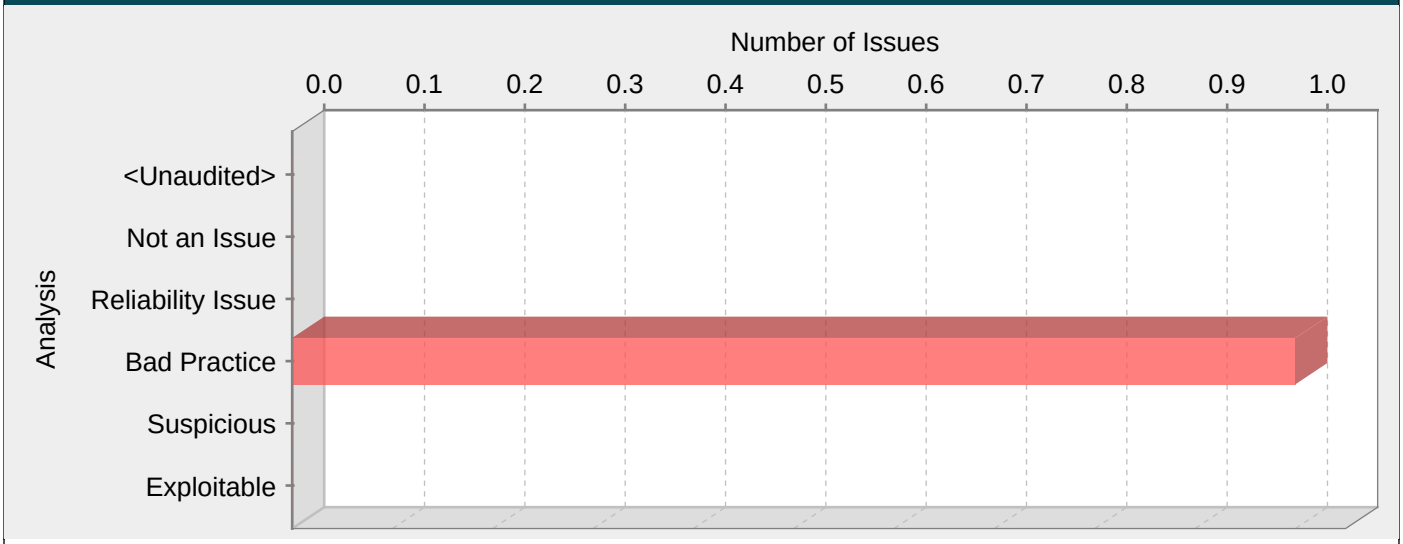
Sink: DataExportController.cs:111 Newtonsoft.Json.Linq.JObject.Parse()

```
109                         var result = client.DownloadString(
110                             $"https://www.google.com/recaptcha/api/siteverify?secret={secretKey}&response={respons
                            e}&remoteip={userIp}");
111                         var obj = JObject.Parse(result);
112                         var status = (bool)obj.SelectToken("success");
113                         return status;
```

Analysis: Not an Issue

Comments:	*dlowe@utah.gov 2018-07-31 12:21 PM*	This response comes directly from google and is not exposed to the front end.

Category: Privacy Violation (1 Issues)



Abstract:

The method HandleCmd() in FTPSClient.cs mishandles confidential information, which can compromise user privacy and is often illegal.

Explanation:

Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system or network.

Example: The following code contains a logging statement that tracks the contents of records added to a database by storing them in a log file. Among other values that are stored, the getPassword() function returns the user-supplied plaintext password associated with the account.

```
pass = GetPassword();
...
dbmsLog.WriteLine(id+":"+pass+":"+type+":"+tstamp);
```

The code in the example above logs a plaintext password to the filesystem. Although many developers trust the filesystem as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer e-mail addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency.

Recommendations:

When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

Tips:

1. As part of any thorough audit for privacy violations, ensure that custom rules have been written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. ASP.NET Request Validation and WCF are among them. To highlight the unvalidated sources of input, the HPE Security Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HPE Security Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. In case of ASP.NET Request Validation, we also provide evidence for when validation is explicitly disabled. We refer to this feature as Context-Sensitive Ranking. To further assist the HPE Security Fortify user with the auditing process, the HPE Security Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

FTPSClient.cs, line 1784 (Privacy Violation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The method HandleCmd() in FTPSClient.cs mishandles confidential information, which can compromise user privacy and is often illegal.		

Source: FTPSClient.cs:2021 Read password()

```
2019        private string PassCmd(string password)
2020        {
2021            return HandleCmd("PASS " + password).Message;
2022        }
```

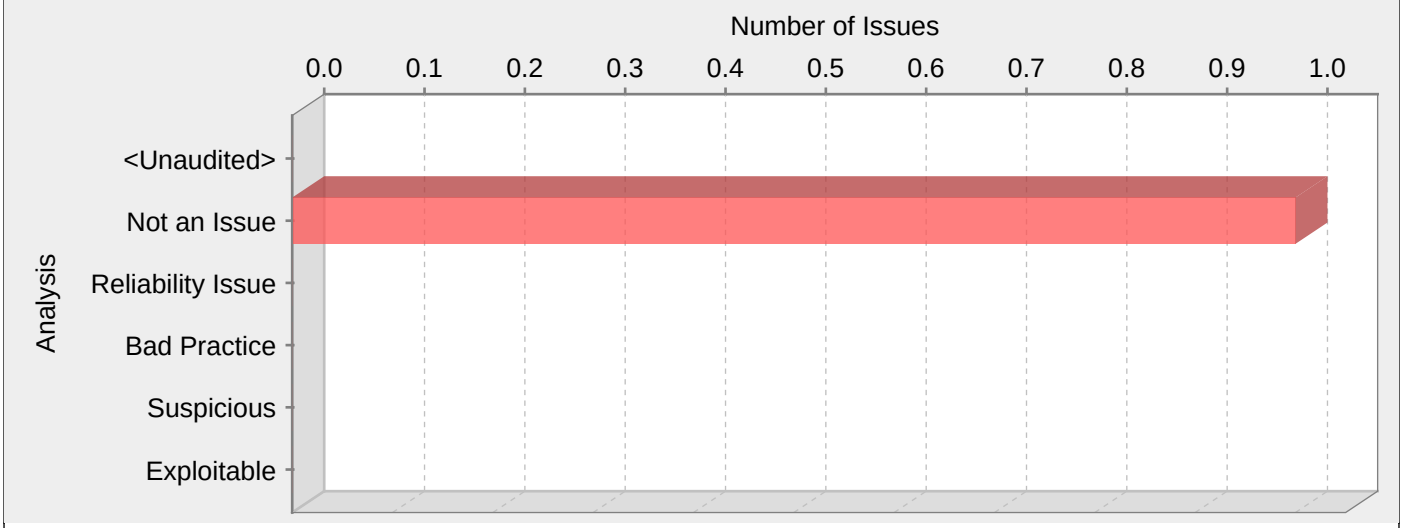
Sink: FTPSClient.cs:1784 System.IO.TextWriter.WriteLine()

```
1782            //    throw new FTPException("Cannot issue a new command while waiting for
a previous one to complete");
1783
1784            ctrlSw.WriteLine(command);
1785            ctrlSw.Flush();
```

Analysis: Bad Practice

Comments:	*dlowe@utah.gov 2018-07-31 12:51 PM*	This code was downloaded through Microsoft NUGET. The program that uses ftps is a scheduled task command line tool. This runs behind a firewall and is not accessible to the outside world. Also the program does not use this specific function.

Category: SQL Injection (1 Issues)



Abstract:

On line 329 of Program.cs, the method UpdateMigrationsTable() invokes a SQL query built using input coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where owner matches the user name of the currently-authenticated user.

```
...
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

The query that this code intends to execute follows:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT * FROM items;
```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name wiley enters the string "name'); DELETE FROM items; --" for itemName, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4]. In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a", the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped meta-characters
- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Recommendations:

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but when user-supplied data needs to be included, they create bind parameters, which are placeholders for data that is subsequently inserted. Bind parameters allow the program to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for the value of each of the bind parameters, without the risk of the data being interpreted as commands.

The previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
...
string userName = ctx.getAuthenticatedUserName();
conn = new SqlConnection(_ConnectionString);
conn.Open();
SqlCommand query = new SqlCommand(
"SELECT * FROM items WHERE itemname=@ItemName
AND owner=@OwnerName", conn);
query.Parameters.AddWithValue("@ItemName", ItemName.Text);
```

query.Parameters.AddWithValue("@OwnerName", userName);
SqlDataReader objReader = objCommand.ExecuteReader();
...

More complicated scenarios, often found in report generation code, require that user input affect the command structure of the SQL statement, such as the addition of dynamic constraints in the WHERE clause. Do not use this requirement to justify concatenating user input into query strings. Prevent SQL injection attacks where user input must affect statement command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Use bind parameters whenever input data must be directly included in a statement.

Tips:

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. ASP.NET Request Validation and WCF are among them. To highlight the unvalidated sources of input, the HPE Security Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HPE Security Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. In case of ASP.NET Request Validation, we also provide evidence for when validation is explicitly disabled. We refer to this feature as Context-Sensitive Ranking. To further assist the HPE Security Fortify user with the auditing process, the HPE Security Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

3. Fortify RTA adds protection against this category.

Program.cs, line 329 (SQL Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 329 of Program.cs, the method UpdateMigrationsTable() invokes a SQL query built using input coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

Source: Program.cs:313 System.Data.SqlClient.SqlCommand.ExecuteReader()

```
311                 sqlconn.Open();
312                 var command = new SqlCommand(sqlQuery, sqlconn);
313                 var reader = command.ExecuteReader();
314                 if (!reader.HasRows)
315                 {
```

Sink: Program.cs:329 System.Data.SqlClient.SqlCommand.set_CommandText()

```
327                             updateCommand.Connection = GetDataBaseConnection();
328                             updateCommand.Connection.Open();
329                             updateCommand.CommandText = "update __MigrationHistory set
MigrationId = '" + newMigrationName +
330                                                         "' where MigrationId = '" +
migrationName + "'";
331                             updateCommand.ExecuteNonQuery();
```

Analysis: Not an Issue

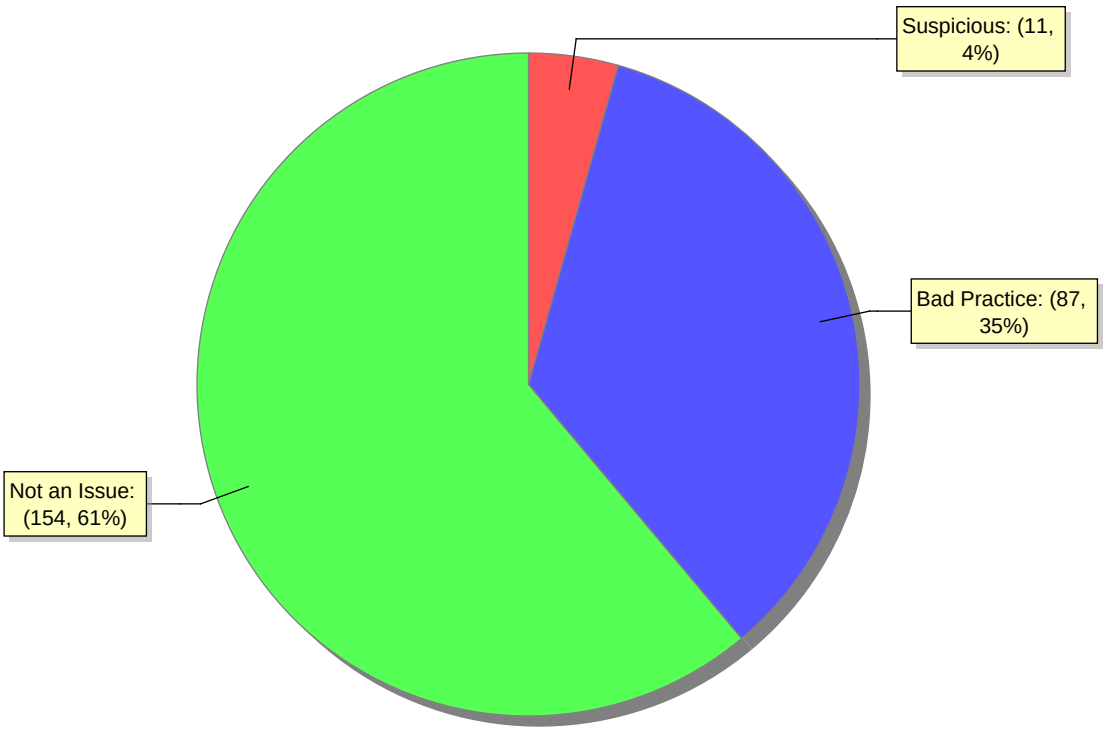
Comments: *dlowe@utah.gov 2018-07-31 12:53 PM* This is reading its input from the database, not from a user.

Issue Count by Category

Issues by Category	
Mass Assignment: Insecure Binder Configuration	50
Password Management: Password in Configuration File	44
ASP.NET MVC Bad Practices: Optional Submodel With Required Property	39
ASP.NET MVC Bad Practices: Controller Action Not Restricted to POST	37
ASP.NET MVC Bad Practices: Model With Optional and Required Properties	30
ASP.NET MVC Bad Practices: Model With Required Non-Nullable Property	23
Path Manipulation	16
Password Management: Hardcoded Password	5
Open Redirect	4
Dangerous Function: xp_cmdshell	1
JSON Injection	1
Privacy Violation	1
SQL Injection	1

Issue Breakdown by Analysis

Issues by Analysis



New Issues

Issues by New Issue

The following issues have been discovered since the last scan.