Fortify Security Report

Feb 6, 2013

112723X

Executive Summary

Issues Overview

On February 06, 2013, a source code review was performed over the MarketStory code base. 77 files, 12060 lines of code were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 441 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	346
Medium	72
High	23

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:/Users/112723X/Desktop/MarketStory compiled V1000 FINAL 2010/MarketStory
Number of Files: 77
Lines of Code: 12060
Build Label: <No Build Label>

Scan Information

Scan time: 01:50
SCA Engine version: 5.10.1.0043
Machine Name: CL6G501
Username running scan: 112723X

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:
$ATTACK_SURFACE$

Filter Set Summary

Current Enabled Filter Set:
Security Auditor 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

Audit Guide Summary

Audit guide not enabled

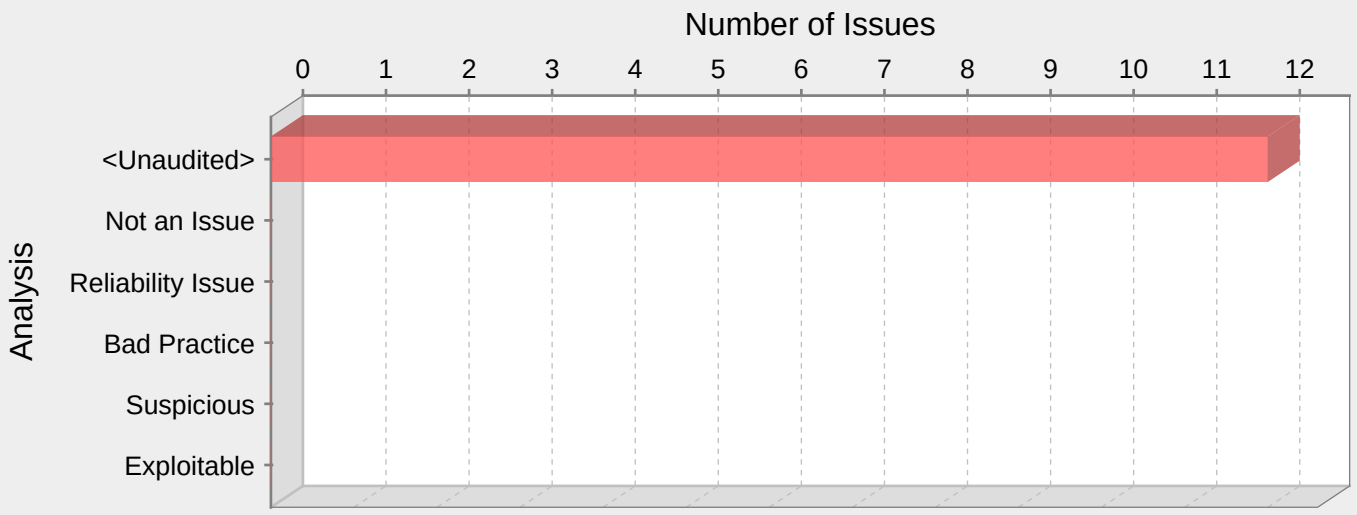
Results Outline

Overall number of results

The scan found 441 issues.

Vulnerability Examples by Category

Category: Insecure Randomness (12 Issues)



Abstract:

Standard pseudo-random number generators cannot withstand cryptographic attacks.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
string GenerateReceiptURL(string baseUrl) {
Random Gen = new Random();
return(baseUrl + Gen.Next().toString() + ".html");
}
```

This code uses the Random.Next() function to generate "unique" identifiers for the receipt pages it generates. Because Random.Next() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Values such as the current time offer only negligible entropy and should not be used.)

The .NET framework provides a cryptographic PRNG in System.Security.Cryptography.RandomNumberGenerator. As is the case with other algorithm-based classes in System.Security, RandomNumberGenerator provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a RandomNumberGenerator object using RandomNumberGenerator.Create(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a RandomNumberGenerator object. If it is unavailable or if you do not specify a particular implementation, then you are given a RandomNumberGenerator implementation selected by the system.

Microsoft provides a single RandomNumberGenerator implementation with the .NET framework named RNGCryptoServiceProvider, which Microsoft describes in the following way:

"To form the seed for the random number generator, a calling application supplies bits it might have-for instance, mouse or keyboard timing input-that are then added to both the stored seed and various system data and user data such as the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block. This result is SHA-1 hashed, and the output is used to seed an RC4 stream, which is then used as the random stream and used to update the stored seed."

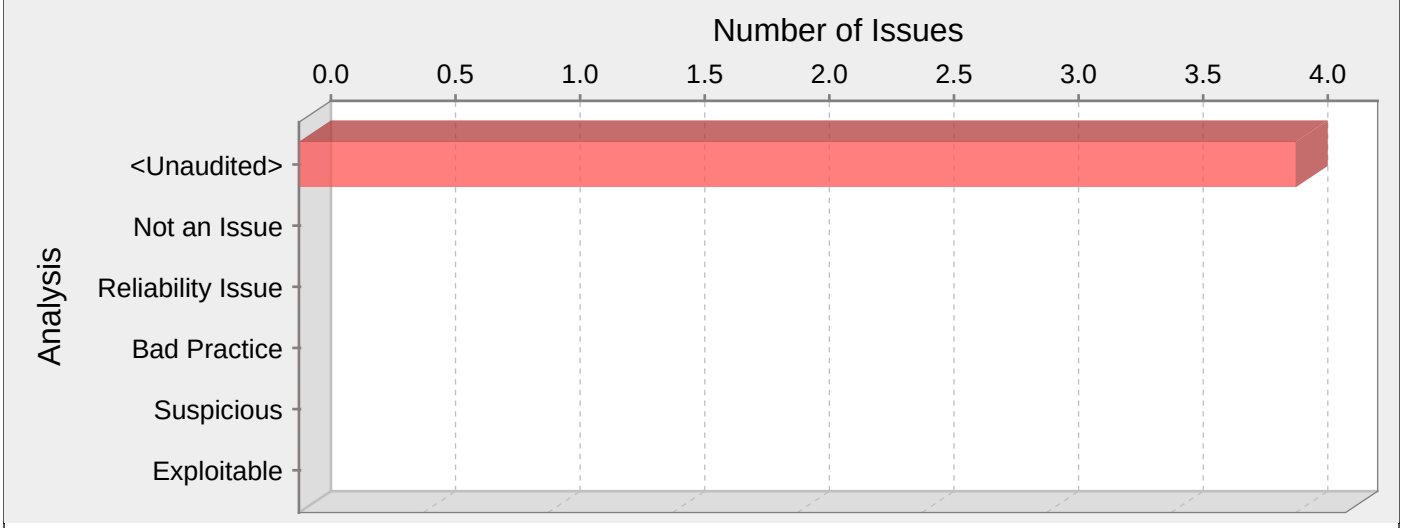
However, the specifics of the Microsoft implementation of the RNGCryptoServiceProvider algorithm are poorly documented, and it is unclear which sources of entropy the implementation uses under what circumstances and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Microsoft implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

AccountActivation.aspx.cs, line 273 (Insecure Randomness)

Fortify Priority:		**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by Next() cannot withstand a cryptographic attack.		
Sink:	AccountActivation.aspx.cs:273 Next()		

```
270
271                 Random rand = new Random((int)DateTime.Now.Ticks);
272                 int RandomNumber;
273                 RandomNumber = rand.Next(100000, 999999);
274                 updateCode(userID, RandomNumber);
275                 string newCode = RandomNumber.ToString();
```

Category: Password Management: Empty Password (4 Issues)



Abstract:

Empty passwords can compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to assign an empty string to a password variable. If the empty password is used to successfully authenticate against another system, then the corresponding account's security is likely compromised because it accepts an empty password. If the empty password is merely a placeholder until a legitimate value can be assigned to the variable, then it can confuse anyone unfamiliar with the code and potentially cause problems on unexpected control flow paths.

Example 1:

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

If the code in Example 1 succeeds, it indicates that the network credential login "scott" is configured with an empty password, which can be easily guessed by an attacker. Even worse, once the program has shipped, updating the account to use a non-empty password will require a code change.

Example 2: The code below initializes a password variable to an empty string, attempts to read a stored value for the password, and compares it against a user-supplied value.

```
...
string storedPassword = "";
string temp;
if ((temp = ReadPassword(storedPassword)) != null) {
storedPassword = temp;
}
if(storedPassword.Equals(userPassword))
// Access protected resources
...
}
...
```

If readPassword() fails to retrieve the stored password due to a database error or another problem, then an attacker could trivially bypass the password check by providing an empty string for userPassword.

Recommendations:

Always read stored password values from encrypted, external resources and assign password variables meaningful values. Ensure that sensitive resources are never protected with empty or null passwords.

Starting with Microsoft(R) Windows(R) 2000, Microsoft(R) provides Windows Data Protection Application Programming Interface (DPAPI), which is an OS-level service that protects sensitive application data, such as passwords and private keys [1].

MainPage.aspx.cs, line 45 (Password Management: Empty Password)

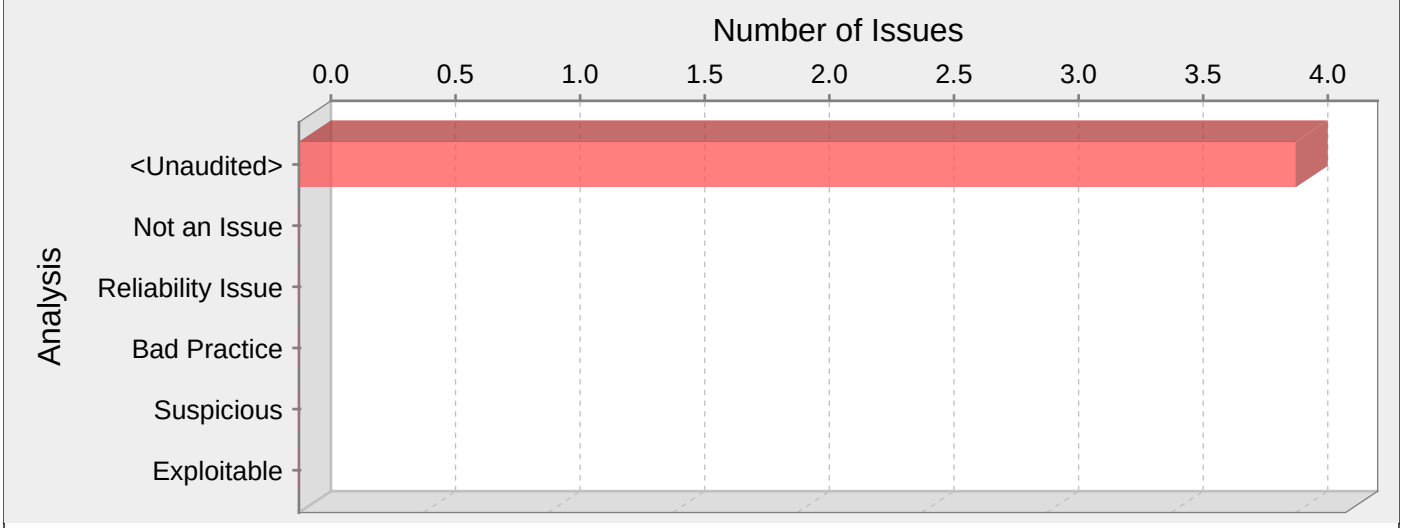
Fortify Priority:	Folder	High

Kingdom:	Security Features
Abstract:	Empty passwords can compromise system security in a way that cannot be easily remedied.

Sink: MainPage.aspx.cs:45 AssignmentStatement()

```
42
43                      MySqlDataReader dr = cmd.ExecuteReader();
44
45                      string password = "";
46
47                      try
48                      {
```

Category: Unreleased Resource: Database (4 Issues)



Abstract:

The program can potentially fail to release a system resource.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool.

Example 1: The following method never closes the file handle it opens. The Finalize() method for StreamReader eventually calls Close(), but there is no guarantee as to how long it will take before the Finalize() method is invoked. In fact, there is no guarantee that Finalize() will ever be invoked. In a busy environment, this can result in the VM using up all of its available file handles.

```
private void processFile(string fName) {
StreamWriter sw = new StreamWriter(fName);
string line;
while ((line = sr.ReadLine()) != null)
processLine(line);
}
```

Example 2: Under normal conditions the following code executes a database query, processes the results returned by the database, and closes the allocated SqlConnection object. But if an exception occurs while executing the SQL or processing the results, the SqlConnection object is not closed. If this happens often enough, the database will run out of available cursors and not be able to execute any more SQL queries.

```
...
SqlConnection conn = new SqlConnection(connString);
SqlCommand cmd = new SqlCommand(queryString);
cmd.Connection = conn;
conn.Open();
SqlDataReader rdr = cmd.ExecuteReader();
HarvestResults(rdr);
conn.Connection.Close();
...
```

Recommendations:

Never rely on Finalize() to reclaim resources. In order for an object's Finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the VM is low on memory, there is no guarantee that an object's Finalize() method will be invoked in an expedient fashion, if it is ever invoked at all (the language does not guarantee that it will be). When the garbage collector finally does run, it can cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. The effect becomes more pronounced as the load on the system increases.

Instead of explicitly closing objects that manage resources, use the C# keyword 'using', which employs the IDisposable interface to perform a cleanup. The following two blocks of code achieve the same result:

The following code uses the finally keyword:

```
StreamReader sr;
try {
sr = new StreamReader(myFileStream);
doWork(sr);
} finally {
if (sr != null) {
sr.Close();
}
}
```

The following code uses the using keyword:

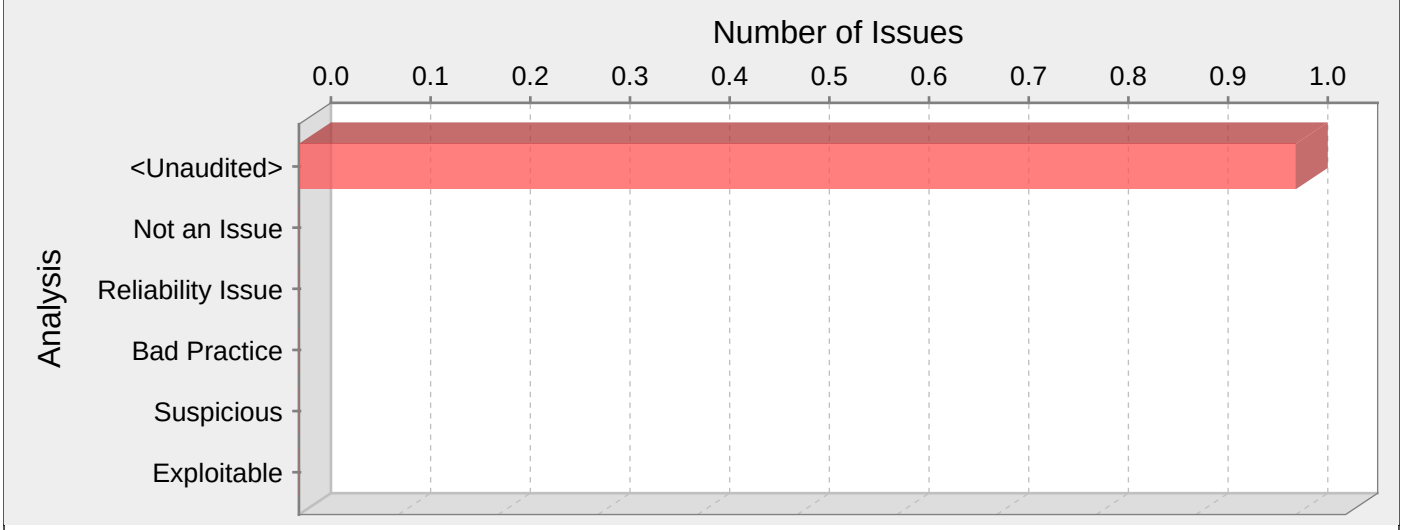
```
using (StreamReader sr = new StreamReader(myFileStream)) {
doWork(sr);
}
```

AutoComplete.asmx.cs, line 47 (Unreleased Resource: Database)

Fortify Priority:		**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function GetCompletionList() in AutoComplete.asmx.cs sometimes fails to release a system resource allocated by MySqlConnection() on line 45.		
Sink:	AutoComplete.asmx.cs:47 conn.Open()		

```
44                  string Query = "SELECT username FROM users WHERE username LIKE '%'
@username '%' AND userID not in (@userID, 1)";
45                  MySqlConnection conn = new MySqlConnection(connectionString);
46                  MySqlCommand cmd = new MySqlCommand(Query, conn);
47                  conn.Open();
48
49                  cmd.Parameters.AddWithValue("@username", prefixText);
50                  cmd.Parameters.AddWithValue("@userID", Session["userID"]);
```

Category: ASP.NET Bad Practices: Non-Serializable Object Stored in Session (1 Issues)



Abstract:

Storing a non-serializable object as an HttpSessionState attribute can damage application reliability.

Explanation:

By default, ASP.NET servers store the HttpSessionState object, its attributes and any objects they reference in memory. This model limits active session state to what can be accommodated by the system memory of a single machine. In order to expand capacity beyond these limitations, servers are frequently configured to persistent session state information, which both expands capacity and permits the replication across multiple machines to improve overall performance. In order to persist its session state, the server must serialize the HttpSessionState object, which requires that all objects stored in it be serializable.

In order for the session to be serialized correctly, all objects the application stores as session attributes must declare the [Serializable] attribute. Additionally, if the object requires custom serialization methods, it must also implement the ISerializable interface.

Example 1: The following class adds itself to the session, but since it is not serializable, the session cannot be serialized correctly.

```
public class DataGlob {
String GlobName;
String GlobValue;

public void AddToSession(HttpSessionState session) {
session["glob"] = this;
}
}
```

Recommendations:

In many cases, the easiest way to fix this problem is to have the offending object declare the [Serializable] attribute. If the class requires custom serialization and deserialization methods, it must also implement the ISerializable interface.

Example 2: The code in Example 1 could be rewritten in the following way:

```
using System.Web;

namespace glob{

[Serializable]
public class DataGlob {
String GlobName;
String GlobValue;

public void AddToSession(HttpSessionState session) {
session["glob"] = this;
}
}

}
```

Generally, implementing a serializable class is straightforward. However, some types of objects will require special treatment. Watch out for objects that hold references to external resources, such as streams and pointers, which are likely to cause complications.

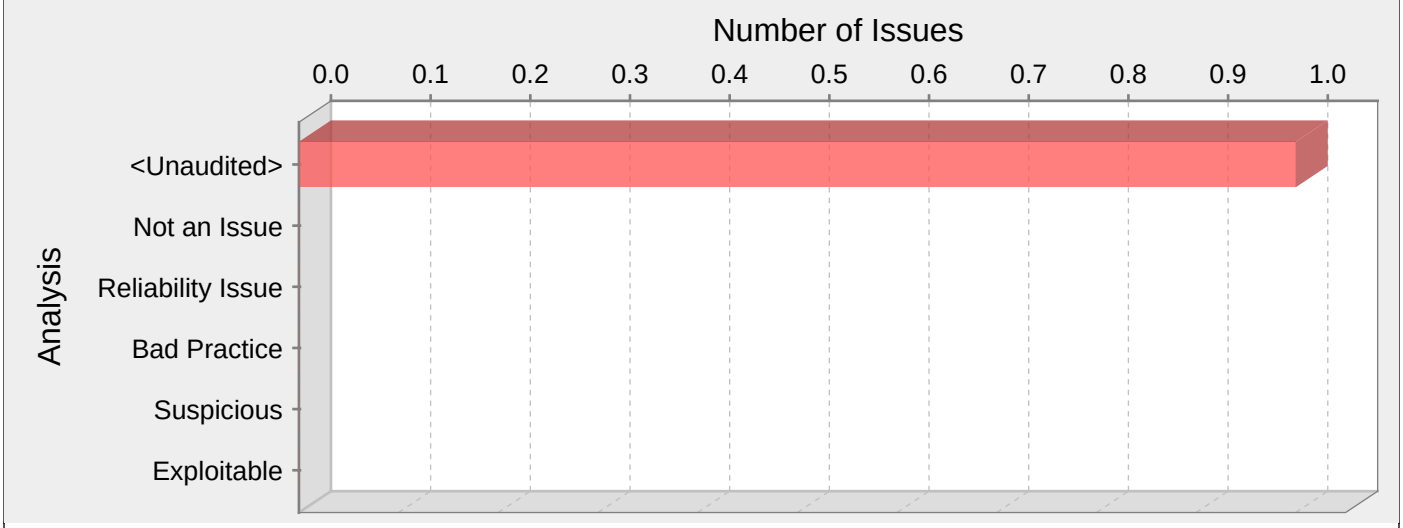
Note that for complex objects, the transitive closure of the objects stored in the session must be serializable. For example, if object A references object B and object A is stored in the session, then both A and B must declare the [Serializable] attribute and implement any necessary serialization or deserialization methods.

GenerateCashCard.aspx.cs, line 99 (ASP.NET Bad Practices: Non-Serializable Object Stored in Session)

Fortify Priority:		**Folder**	High
Kingdom:	Time and State		
Abstract:	The method Button1_Click() in GenerateCashCard.aspx.cs stores a non-serializable object as an HttpSessionState attribute on line 99, which can damage application reliability.		
Sink:	GenerateCashCard.aspx.cs:99 FunctionCall: set_Item()		

```
96                      row.Controls.Add(securityCodeCell);
97                      Table1.Controls.Add(row);
98
99                      Session["TableContent"] = Table1;
100                     da.adminCardLog(quantity, cashValue, userid);
101                 }
102             }
```

Category: Cross-Site Scripting: Reflected (1 Issues)



Abstract:

Sending unvalidated data to a web browser can result in the browser executing malicious code.

Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Reflected XSS, the untrusted source is typically a web request, while in the case of Persisted (also known as Stored) XSS it is typically a database or other back-end datastore.

2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following ASP.NET Web Form reads an employee ID number from an HTTP request and displays it to the user.

```
<script runat="server">
...
EmployeeID.Text = Login.Text;
...
</script>
```

Where Login and EmployeeID are form controls defined as follows:

```
<form runat="server">
<asp:TextBox runat="server" id="Login"/>
...
<asp:Label runat="server" id="EmployeeID"/>
</form>
```

Example 2: The following ASP.NET code segment shows the programmatic way to implement Example 1 above.

```
protected System.Web.UI.WebControls.TextBox Login;
protected System.Web.UI.WebControls.Label EmployeeID;
...
EmployeeID.Text = Login.Text;
```

The code in these examples operates correctly if Login contains only standard alphanumeric text. If Login has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks in order to lure victims into clicking a link. When the victims click the link, they unwittingly reflect the malicious content through the vulnerable web application and back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 3: The following ASP.NET Web Form queries a database for an employee with a given employee ID and prints the name corresponding with the ID.

```
<script runat="server">
...
string query = "select * from emp where id=" + eid;
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
string name = dt.Rows[0]["Name"];
...
EmployeeName.Text = name;
</script>
```

Where EmployeeName is a form control defined as follows:

```
<form runat="server">
...
<asp:Label id="EmployeeName" runat="server">
...
</form>
```

Example 4: The following ASP.NET code segment is functionally equivalent to Example 3 above, but implements all of the form elements programmatically.

```
protected System.Web.UI.WebControls.Label EmployeeName;
...
string query = "select * from emp where id=" + eid;
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
string name = dt.Rows[0]["Name"];
...
EmployeeName.Text = name;
```

As in Examples 1 and 2, these code examples function correctly when the values of name are well-behaved, but they nothing to prevent exploits if the values are not. Again, these can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Examples 1 and 2, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Examples 3 and 4, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

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 rulepacks dynamically re-prioritize the issues reported by HP 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 HP Fortify user with the auditing process, the Fortify 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.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server -side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- The semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

The HP Fortify Secure Coding Rulepacks treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources.

Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against Cross-Site Scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

Fortify RTA adds protection against this category.

SearchResults.aspx.cs, line 18 (Cross-Site Scripting: Reflected)

Fortify Priority:		**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method Page_Load() in SearchResults.aspx.cs sends unvalidated data to a web browser on line 18, which can result in the browser executing malicious code.		

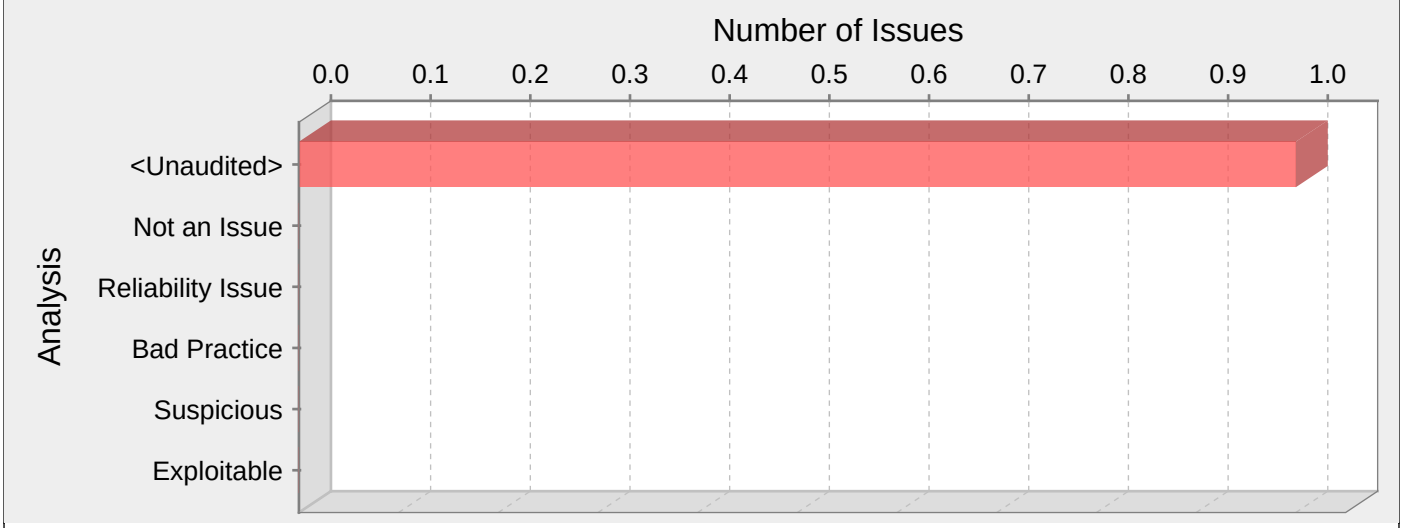
Source: SearchResults.aspx.cs:18 System.Web.HttpRequest.get_QueryString()

```
15          {
16              if (Session["userID"] != null)
17              {
18                  Label1.Text = Request.QueryString["value"];
19
20                  if (Request.QueryString["type"] == "Users")
21                  {
```

Sink: SearchResults.aspx.cs:18 System.Web.UI.WebControls.Label.set_Text()

```
15          {
16              if (Session["userID"] != null)
17              {
18                  Label1.Text = Request.QueryString["value"];
19
20                  if (Request.QueryString["type"] == "Users")
21                  {
```

Category: Header Manipulation: Cookies (1 Issues)



Abstract:

Including unvalidated data in an HTTP response header can enable cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

Explanation:

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers and frameworks will prevent the injection of malicious characters into HTTP headers. For example, recent versions of Microsoft's .NET framework will convert CR, LF, and NULL characters to %0d, %0a and %00 when they are sent to the HttpResponse.AddHeader() method. If you are using the latest .NET framework that prevents setting headers with new line characters, then your application might not be vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

Example: The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

```
protected System.Web.UI.WebControls.TextBox Author;
...
string author = Author.Text;
Cookie cookie = new Cookie("author", author);
...
```

Assuming a string consisting of standard alpha-numeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for Author.Text does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
...
```

Set-Cookie: author=Wiley Hacker

HTTP/1.1 200 OK

...

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting and page hijacking.

Cross-User Defacement: An attacker can make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker can leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Cache Poisoning: The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

Cross-Site Scripting: Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Page Hijacking: In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker can cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Cookie Manipulation: When combined with attacks like Cross-Site Request Forgery, attackers can change, add to, or even overwrite a legitimate user's cookies.

Open Redirect: Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Recommendations:

The solution to Header Manipulation is to ensure that input validation occurs in the correct places and checks for the correct properties.

Since Header Manipulation vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create a whitelist of safe characters that are allowed to appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alpha-numeric characters or an account number might only include digits 0-9.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack, other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

Once you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

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 HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP 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 HP Fortify user with the auditing process, the Fortify 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.

Fortify RTA adds protection against this category.

Site.Master.cs, line 42 (Header Manipulation: Cookies)

Fortify Priority:	**Folder** High
Kingdom:	Input Validation and Representation
Abstract:	The method Page_Init() in Site.Master.cs includes unvalidated data in an HTTP response header on line 42. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

Source: Site.Master.cs:23 System.Web.HttpRequest.get_Cookies()

```
20          protected void Page_Init(object sender, EventArgs e)
21          {
22              // The code below helps to protect against XSRF attacks
23              var requestCookie = Request.Cookies[AntiXsrfTokenKey];
24              Guid requestCookieGuidValue;
25
26              if (requestCookie != null && Guid.TryParse(requestCookie.Value, out
requestCookieGuidValue))
```

Sink: Site.Master.cs:42 System.Web.HttpCookie.set_Value()

```
39                  {
40                      HttpOnly = true,
41                      Value = _antiXsrfTokenValue
42                  };
43                  if (FormsAuthentication.RequireSSL && Request.IsSecureConnection)
44                  {
45                      responseCookie.Secure = true;
```

Issue Count by Category

Issues by Category	
Poor Error Handling: Overly Broad Catch	134
Poor Error Handling: Empty Catch Block	119
Cross-Site Scripting: Poor Validation	73
Missing Check against Null	25
Dead Code: Unused Field	20
Insecure Randomness	12
JavaScript Hijacking: Vulnerable Framework	9
Trust Boundary Violation	8
SQL Injection	7
Password Management: Password in Comment	6
Password Management: Empty Password	4
Dead Code: Unused Method	4
Unreleased Resource: Database	4
Resource Injection	4
Password Management: Null Password	2
Cross-Site Request Forgery	2
Weak Cryptographic Hash	1
Cross-Site Scripting: Reflected	1
Cookie Security: Cookie not Sent Over SSL	1
ASP.NET Misconfiguration: Debug Information	1
Header Manipulation: Cookies	1
Password Management	1
ASP.NET Bad Practices: Non-Serializable Object Stored in Session	1
JavaScript Hijacking: Ad Hoc Ajax	1

Issue Breakdown by Analysis

Issues by Analysis