Fortify Security Report

7 Feb 2022

Demo User

Executive Summary

Issues Overview

On 7 Feb 2022, a source code review was performed over the JavaARMDemo code base. 6 files, 86 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 35 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	21
High	9
Critical	4
Medium	1

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/klee/source/repos/JavaARMDemo
Number of Files: 6
Lines of Code: 86
Build Label: SNAPSHOT

Scan Information

Scan time: 00:15
SCA Engine version: 21.2.2.0004
Machine Name: GBklee01
Username running scan: klee

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:
Environment Variables:
java.lang.System.getenv

Private Information:
null.null.null
java.lang.System.getenv

System Information:
null.null.null

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

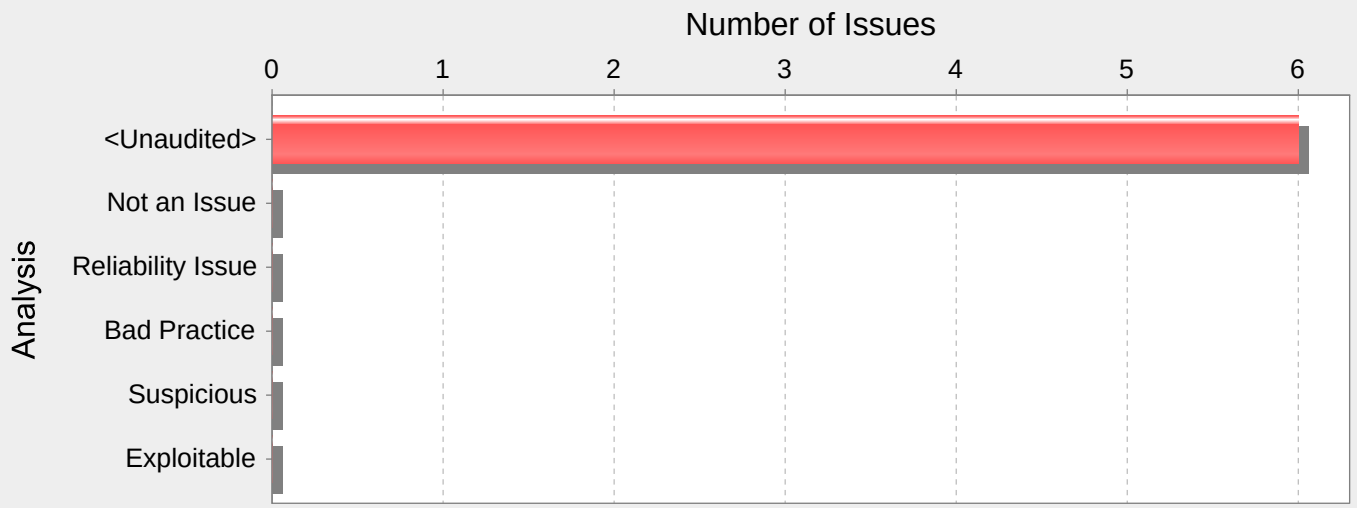
Results Outline

Overall number of results

The scan found 35 issues.

Vulnerability Examples by Category

Category: Unreleased Resource: Database (6 Issues)



Abstract:

The function getProducts() in DbAccess.java sometimes fails to release a database resource allocated by getConnection() on line 15.

Explanation:

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. However, if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

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

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
harvestResults(rs);
stmt.close();
```

Recommendations:

1. 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 JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may 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. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for the Example should be rewritten as follows:

```
public void execCxnSql(Connection conn) {
Statement stmt;
try {
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
```

```
...
}
finally {
if (stmt != null) {
safeClose(stmt);
}
}
}

public static void safeClose(Statement stmt) {
if (stmt != null) {
try {
stmt.close();
} catch (SQLException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the statement. Presumably this helper function will be reused whenever a statement needs to be closed.

Also, the execCxnSql method does not initialize the stmt object to null. Instead, it checks to ensure that stmt is not null before calling safeClose(). Without the null check, the Java compiler reports that stmt might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If stmt is initialized to null in a more complex method, cases in which stmt is used without being initialized will not be detected by the compiler.

Tips:

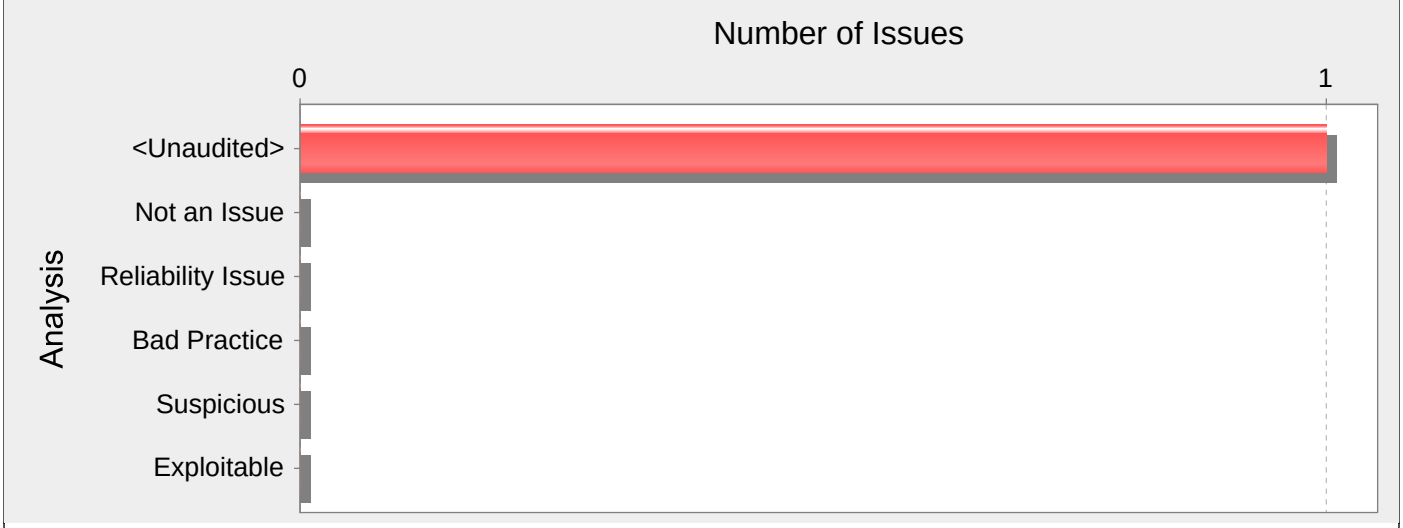
1. Be aware that closing a database connection may or may not automatically free other resources associated with the connection object. If the application uses connection pooling, it is best to explicitly close the other resources after the connection is closed. If the application is not using connection pooling, the other resources are automatically closed when the database connection is closed. In such a case, this vulnerability is invalid.

DbAccess.java, line 15 (Unreleased Resource: Database)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function getProducts() in DbAccess.java sometimes fails to release a database resource allocated by getConnection() on line 15.		
Sink:	DbAccess.java:15 conn = getConnection()		

```
13
14                    public ArrayList<Product> getProducts(String query) throws Exception {
15                            Connection conn = getConnection();
16
17                     initDbIfNeed(conn);
```

Category: Access Control: Azure Storage (1 Issues)



Abstract:

On line 45 of azuredeploy.json, the template defines an Azure storage account with unrestricted network access.

Explanation:

By default, Azure storage accounts accept connections from clients on any network.

Loose access configurations unnecessarily expose systems and broaden an organization's attack surface. Services open to interaction with the public are subjected to almost continuous scanning and probing by malicious entities.

Example 1: The following example template fails to restrict network access to an Azure storage account.

```
{
...
"resources": [
{
"name": "VulnerableStorageAccount",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2019-06-01",
"tags": {
"displayName": "vulnerablestore1"
},
"location": "[resourceGroup().location]",
...
"properties": {
"networkAcls": {
"bypass": "None",
"defaultAction": "Allow"
}
}
}
],
"outputs": {}
}
```

Recommendations:

Restrict critical services access to specific IP addresses, ranges, or virtual networks.

Configure Azure Storage firewall rules to block all traffic by default and specify the particular sources of trusted communication.

To specify trusted sources, configure the networkAcls/ipRules, networkAcls/resourceAccessRules or networkAcls/virtualNetworkRules. Then, to block all traffic by default, configure the networkAcls/defaultAction property to Deny.

Note that the networkAcls property was introduced in apiVersion 2017-06-01 and later.

Example 2: The following example fixes the problem in Example 1. It restricts network access to an Azure storage account to a specific virtual network.

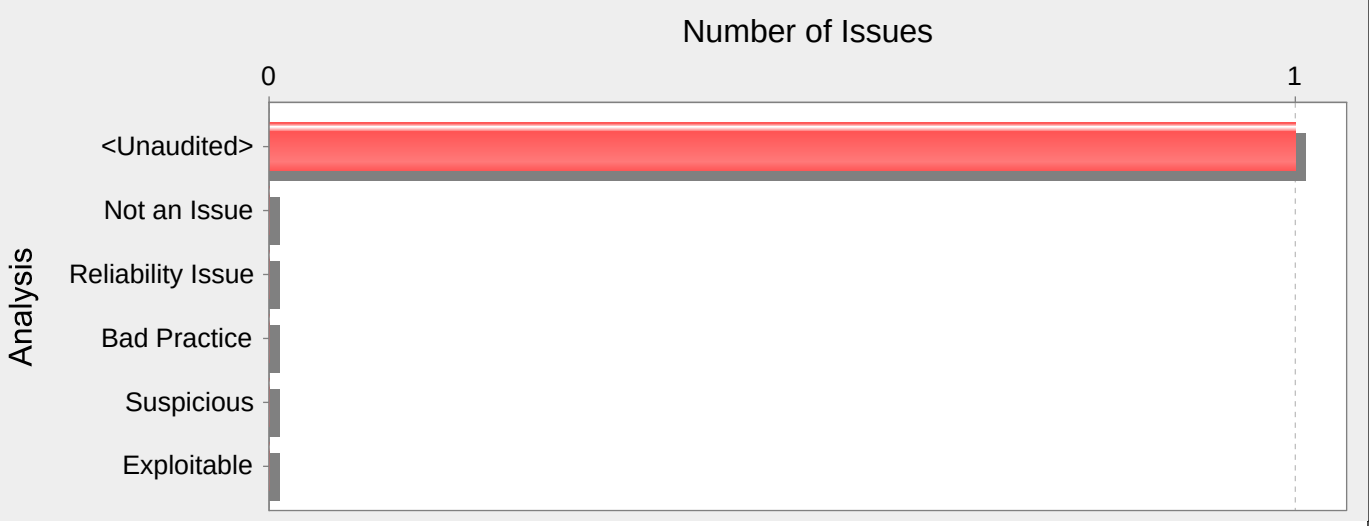
```
{
...
"resources": [
{
"name": "VulnerableStorageAccount",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2019-06-01",
"tags": {
"displayName": "vulnerablestore1"
},
"location": "[resourceGroup().location]",
...
"properties": {
"networkAcls": {
"bypass": "None",
"virtualNetworkRules": [
{
"id": "[variables('subnetId')[0]]",
"action": "Allow"
}
],
"defaultAction": "Deny"
}
}
}
],
"outputs": {}
}
```

azuredeploy.json, line 45 (Access Control: Azure Storage)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 45 of azuredeploy.json, the template defines an Azure storage account with unrestricted network access.		
Sink:	azuredeploy.json:45 ConfigMap()		

```
43          },
44          "resources": [
45              {
46                  "type": "Microsoft.Storage/storageAccounts",
47                  "sku": {
```

Category: Azure Resource Manager Misconfiguration: Insecure Transport (1 Issues)



Abstract:

On line 89 of azuredeploy.json, the template defines an Azure App Service that does not enforce HTTPS communication.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

Serving web applications over HTTP allows attackers to perform man-in-the-middle attacks, giving them access to read or modify the data in transit over the channel.

Example 1: The following example shows a template that defines an Azure App Service that does not enforce HTTPS communication.

```
"resources": [
{
"name": "webSite",
"type": "Microsoft.Web/sites",
"apiVersion": "2020-12-01",
"location": "location1",
"tags": {},
"properties": {
"enabled": true
},
"resources": []
}
```

Recommendations:

Enforce transport encryption to ensure that all data exchange is performed over encrypted channels and inhibit man-in-the-middle attacks.

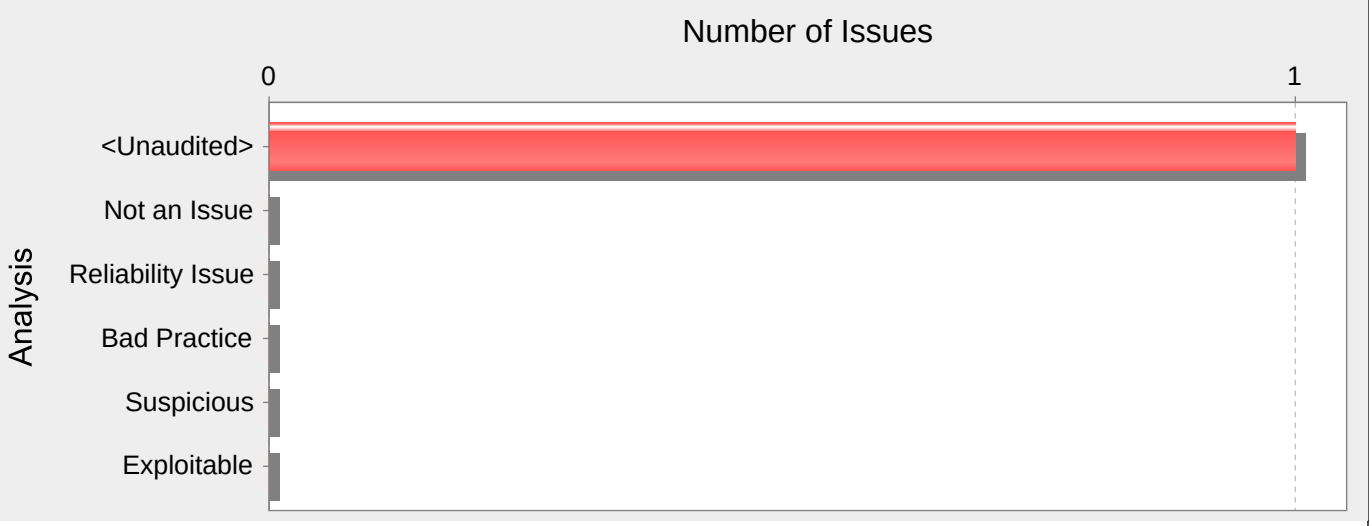
Example 2: The following example fixes the problem in Example 1. It enforces HTTPS communication by enabling the httpsOnly property.

```
"resources": [
{
"name": "webSite",
"type": "Microsoft.Web/sites",
"apiVersion": "2020-12-01",
"location": "location1",
"tags": {},
"properties": {
"enabled": true,
"httpsOnly": true
},
"resources": []
}
```

azuredeploy.json, line 89 (Azure Resource Manager Misconfiguration: Insecure Transport)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Environment		
Abstract:	On line 89 of azuredeploy.json, the template defines an Azure App Service that does not enforce HTTPS communication.		
Sink:	azuredeploy.json:89 ConfigMap()		

```
87                      }
88                  },
89                  {
90                      "type": "Microsoft.Web/sites",
91                      "kind": "app",
```

Category: Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy (1 Issues)



Abstract:

On line 55 of azuredeploy.json, the template defines an overly permissive CORS policy.

Explanation:

Cross-Origin Resource Sharing, commonly referred to as CORS, is a technology that allows a domain to define a policy for its resources to be accessed by a web page hosted on a different domain. Historically, web browsers have restricted their domain resources from being accessed by scripts loaded from a different domain to abide by the same origin policy.

CORS provides a method for a domain to whitelist other domains and allow them to access its resources.

Be careful when defining a CORS policy because an overly permissive policy configured at the server level for a domain or a directory on a domain can expose more content for cross domain access than intended. CORS can enable a malicious application to communicate with a victim application inappropriately, which can lead to information disclosure, spoofing, data theft, relay, or other attacks.

Implementing CORS can increase an application's attack surface and should only be used when necessary.

Example 1: The following example template defines an overly permissive CORS policy for an Azure SignalR web application.

```
{
...
"type": "Microsoft.SignalRService/SignalR",
...
"properties": {
...
"cors": {
"allowedOrigins": ["*"]
},
...
}
```

Example 2: The following example template defines an overly permissive CORS policy for an Azure web application.

```
{
"apiVersion": "2020-12-01",
"type": "Microsoft.Web/sites",
...
"properties": {
...
"siteConfig": {
...
"cors": {
"allowedOrigins": [
"*"
]
},
```

```
...
}
```

Example 3: The following example template defines an overly permissive CORS policy for an Azure Maps account.

```
{
"apiVersion": "2021-12-01-preview",
"type": "Microsoft.Maps/accounts",
...
"properties":{
"cors":{
"allowedOrigins": ["*"]
}
},
...
}
```

Example 4: The following example template defines an overly permissive CORS policy for an Azure Cosmos DB account.

```
{
"type": "Microsoft.DocumentDB/databaseAccounts",
...
"properties": {
"cors": [{
"allowedOrigins":"*"
}],
...
}
```

Example 5: The following example template defines an overly permissive CORS policy for an Azure storage blob service.

```
{
"type": "Microsoft.Storage/storageAccounts/blobServices",
...
"properties": {
"cors": {
"corsRules": [
{
"allowedOrigins":["*"],
...
}
...
}
```

Recommendations:

Do not use a wildcard (*) as the value for the allowedOrigins property. Instead, provide an explicit list of trusted domains.

Example 6: The following example fixes the problem in Example 1 by specifying an explicitly trusted domain for an Azure SignalR web application.

```
{
...
"type": "Microsoft.SignalRService/SignalR",
...
"properties": {
...
"cors": {
"allowedOrigins": ["www.trusted.com"]
},
...
}
```

Example 7: The following example fixes the problem in Example 2 by specifying an explicitly trusted domain for an Azure web application.

```
{
"apiVersion": "2020-12-01",
"type": "Microsoft.Web/sites",
...
"properties": {
...
"siteConfig": {
...
"cors": {
"allowedOrigins": [
"www.trusted.com"
]
},
...
}
```

Example 8: The following example fixes the problem in Example 3 by specifying an explicitly trusted domain for an Azure Maps account.

```
{
"apiVersion": "2021-12-01-preview",
"type": "Microsoft.Maps/accounts",
...
"properties":{
"cors":{
"allowedOrigins": ["www.trusted.com"]
}
},
...
}
```

Example 9: The following example fixes the problem in Example 4 by specifying an explicitly trusted domain for an Azure Cosmos DB.

```
{
"type": "Microsoft.DocumentDB/databaseAccounts",
...
"properties": {
"cors": [{
"allowedOrigins":"www.trusted.com"
}],
...
}
```

Example 10: The following example fixes the problem in Example 5 by specifying an explicitly trusted domain for an Azure blob Service.

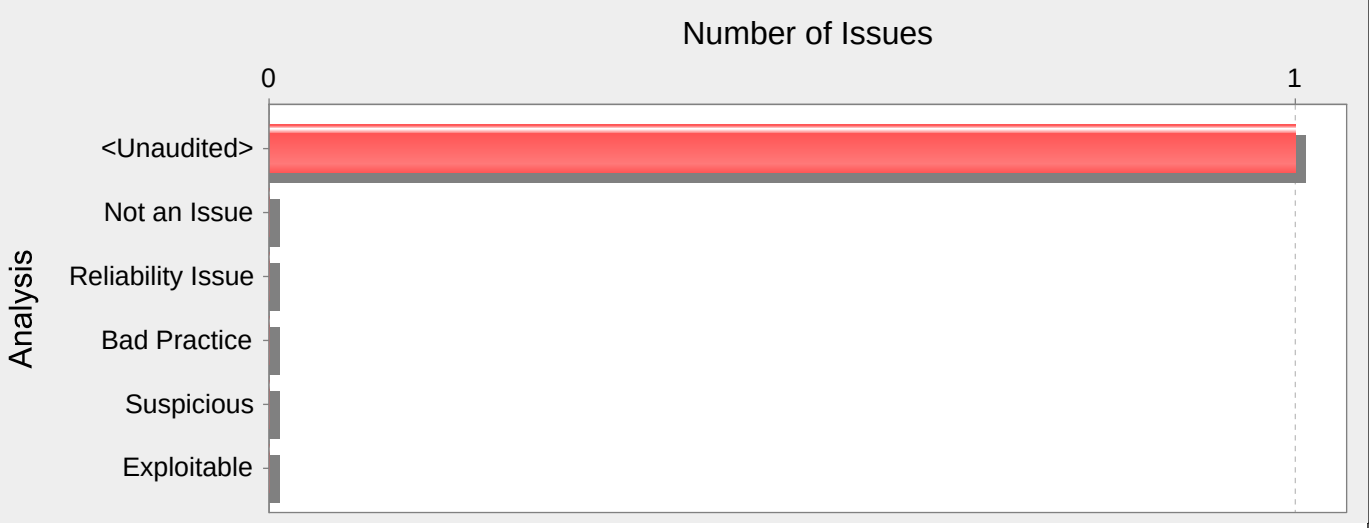
```
{
"type": "Microsoft.Storage/storageAccounts/blobServices",
...
"properties": {
"cors": {
"corsRules": [
{
"allowedOrigins":["www.trusted.com"],
...
}
...
```

```
}
```

azuredeploy.json, line 55 (Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 55 of azuredeploy.json, the template defines an overly permissive CORS policy.		
Sink:	azuredeploy.json:55 ConfigMap()		

```
53                  "location": "[parameters('location')]",
54                  "properties": {
55                  "cors": {
56                    "corsRules": [
57                      {
```

Category: Insecure Transport: Azure Storage (1 Issues)



Abstract:

On line 45 of azuredeploy.json, the template defines a storage account that does not enforce encryption in transit.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

By default, in apiVersion 2019-04-01 and later, Azure storage accounts require a secure connection to respond to requests, but the requirement can be disabled with the supportsHttpsTrafficOnly property.

Disabling transfer security exposes the data to unauthorized access, potential theft, and tampering.

Example 1: The following example template shows a storage account that does not enforce secure transfer.

```
{
"name": "Storage1",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2021-02-01",
...
"properties": {
"supportsHttpsTrafficOnly": false
}
}
```

Recommendations:

Ensure that Azure storage accounts require secure connections from clients by setting supportsHttpsTrafficOnly to true. Note that omitting this setting in templates prior to apiVersion 2019-04-01 defaults to allowing insecure connections to Azure storage accounts.

Example 2: The following example fixes the problem in Example 1. It explicitly configures the Azure storage account to only accept requests over a secure channel.

```
{
"name": "Storage1",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2021-02-01",
...
"properties": {
"supportsHttpsTrafficOnly": true
}
}
```

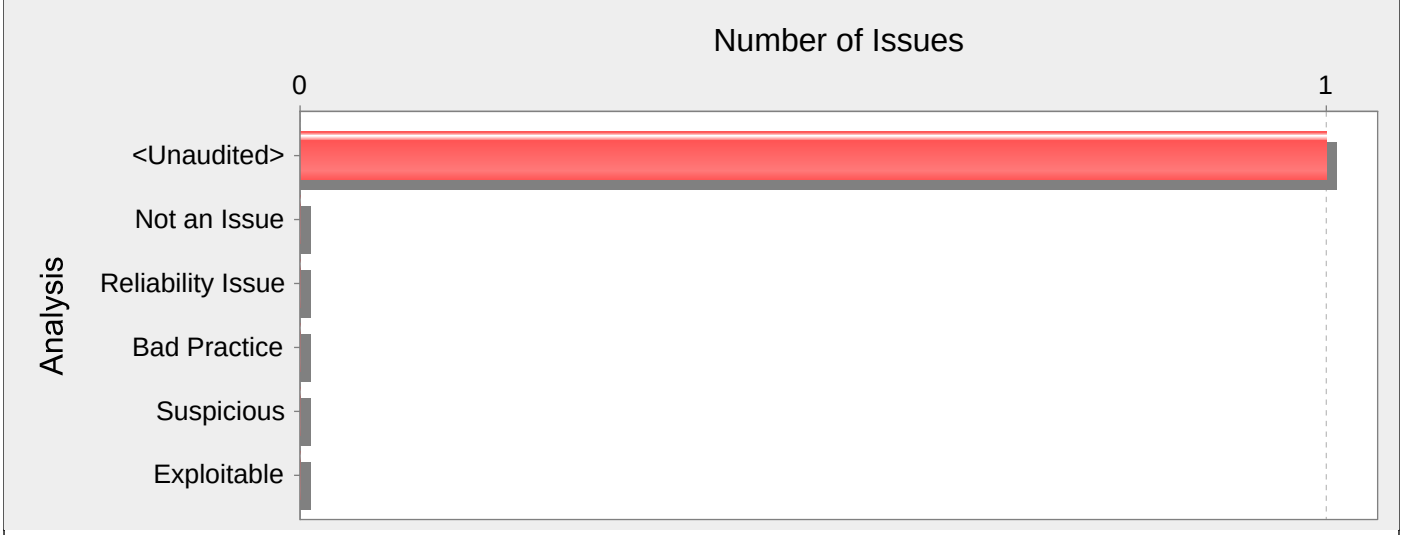
azuredeploy.json, line 45 (Insecure Transport: Azure Storage)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 45 of azuredeploy.json, the template defines a storage account that does not enforce encryption in transit.		

Sink: azuredeploy.json:45 ConfigMap()

```
43                      },
44                      "resources": [
45                          {
46                              "type": "Microsoft.Storage/storageAccounts",
47                              "sku": {
```

Category: Insecure Transport: Database (1 Issues)



Abstract:

On line 146 of azuredeploy.json, the template explicitly disables database transport encryption.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

Disabling transport security exposes the data to unauthorized access, potential theft, and tampering.
By default, Azure MySQL Database enables transport encryption, but it can be disabled with the sslEnforcement property.

Example 1: The following example template defines an Azure MySQL Database with transport encryption disabled.

```
{
"type": "Microsoft.DBforMySQL/servers",
"apiVersion": "2017-12-01",
"name": "[parameters('serverName')]",
"location": "[parameters('location')]",
"sku": {
...
},
"properties": {
"createMode": "Default",
"version": "[parameters('mysqlqlVersion')]",
...
"sslEnforcement": "Disabled",
...
}
```

Recommendations:

Ensure that database transport encryption is enabled to protect data in transit.
Example 2: The following example fixes the problem in Example 1. It explicitly enables transport encryption.

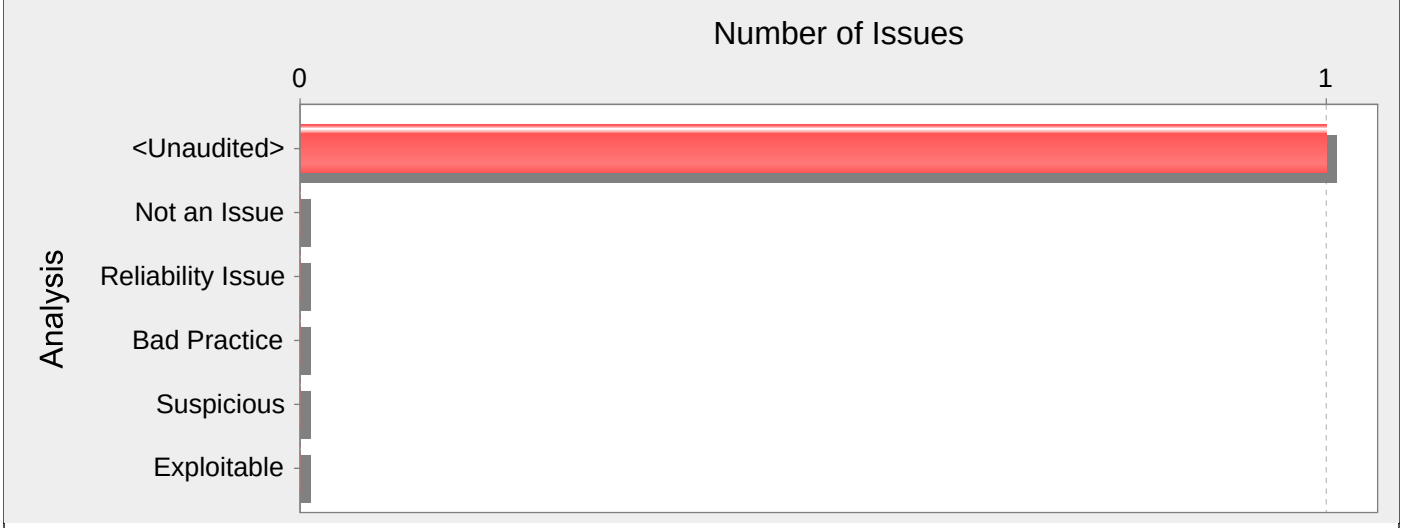
```
{
"type": "Microsoft.DBforMySQL/servers",
"apiVersion": "2017-12-01",
"name": "[parameters('serverName')]",
"location": "[parameters('location')]",
"sku": {
...
},
"properties": {
"createMode": "Default",
"version": "[parameters('mysqlqlVersion')]",
...
```

"sslEnforcement": "Enabled",
...
}

azuredeploy.json, line 146 (Insecure Transport: Database)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 146 of azuredeploy.json, the template explicitly disables database transport encryption.		
Sink:	azuredeploy.json:146 ConfigMap()		

```
144                 }
145             },
146             {
147                 "type": "Microsoft.DBforMySQL/servers",
148                 "apiVersion": "2017-12-01",
```

Category: Insecure Transport: Weak SSL Protocol (1 Issues)



Abstract:

On line 89 of azuredeploy.json, the template allows the use of outdated TLS versions.

Explanation:

The Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols provide a protection mechanism to ensure the authenticity, confidentiality, and integrity of data transmitted between a client and web server. Both TLS and SSL have undergone revisions that result in periodic version updates. Each new revision addresses security weaknesses discovered in the previous version. Use of an insecure version of TLS/SSL weakens the strength of the data protection and might allow an attacker to compromise, steal, or modify sensitive information.

Insecure versions of TLS/SSL might exhibit one or more of the following properties:

- No protection against man-in-the-middle attacks
- Same key used for authentication and encryption
- Weak message authentication control
- No protection against TCP connection closing

The presence of these properties might enable an attacker to intercept, modify, or tamper with sensitive data.

Example 1: The following example template shows an app with the minTlsVersion set to an outdated version of the TLS protocol.

```
{
...
"apiVersion": "2020-06-01",
"name": "[parameters('webAppName')]",
"location": "[parameters('location')]",
"properties": {
"siteConfig": {
"minTlsVersion": "1.0",
"linuxFxVersion": "[parameters('linuxFxVersion')]"
},
...
}
```

Recommendations:

It is highly recommended to force the application to use only the most secure protocols.

Please note that some clients might not support the newer ciphers. It is essential for an organization to upgrade those clients and discontinue the use of outdated and insecure ciphers.

Ensure that Azure App Services enforces a secure minimum TLS version. This can either be configured for a specific app or for all apps within a certain App Services Environment (ASE).

To disable TLS versions 1.0 and 1.1 on a specific app, set the Microsoft.Web/sites/config/minTlsVersion property to the latest version that Azure App Services supports. This can either be configured for a specific app or for all apps within a certain App Services Environment (ASE).

Example 2: The following example template fixes the issue with Example 1.

```
{
```

...
"apiVersion": "2020-06-01",
"name": "[parameters('webAppName')]",
"location": "[parameters('location')]",
"properties": {
"siteConfig": {
"minTlsVersion": "1.2",
"linuxFxVersion": "[parameters('linuxFxVersion')]"
},
...
}

Example 3: To disable TLS versions 1.0 and 1.1 on an ASE, use Microsoft.Web/HostingEnvironments/clusterSettings.

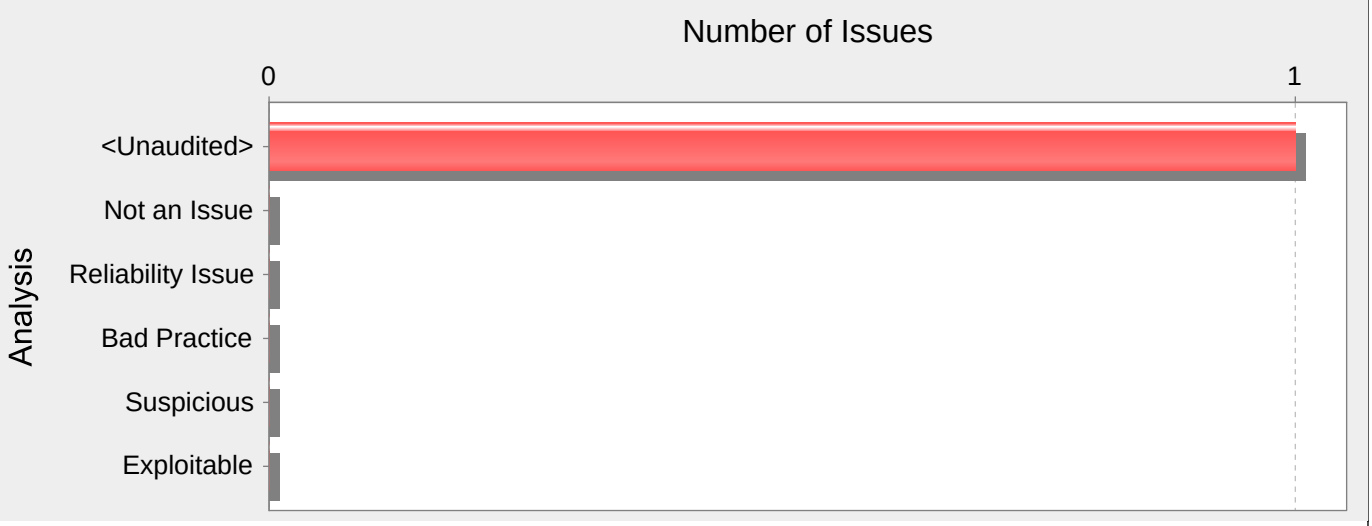
"clusterSettings": [
{
"name": "DisableTls1.0",
"value": "1"
}

azuredeploy.json, line 89 (Insecure Transport: Weak SSL Protocol)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 89 of azuredeploy.json, the template allows the use of outdated TLS versions.		
Sink:	azuredeploy.json:89 ConfigMap()		

```
87                      }
88                  },
89                  {
90                      "type": "Microsoft.Web/sites",
91                      "kind": "app",
```

Category: Password Management: Hardcoded Password (1 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is difficult to remedy.

Explanation:

Never hardcode passwords. Not only does it expose the password to all of the project's developers, it also makes fixing the problem extremely difficult. After the code is in production, a program patch is probably the only way to change the password. If the account the password protects is compromised, the system owners must choose between security and availability.

Example: The following JSON uses a hardcoded password:

...
{
"username":"scott"
"password":"tiger"
}
...

This configuration may be valid, but anyone who has access to the configuration will have access to the password. After the program is released, changing the default user account "scott" with a password of "tiger" is difficult. Anyone with access to this information can use it to break into the system.

Recommendations:

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

azuredeploy.json, line 156 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is difficult to remedy.		

Sink: azuredeploy.json:156 ConfigPair()

```
154                    "storageMB": 51200,
155                    "administratorLogin": "[parameters('mySqlAdminLogin')]",
156                    "administratorLoginPassword": "[parameters('mySqlAdminPassword')]",
157                    "sslEnforcement" : "Disabled",
158                    "backupRetentionDays": "7",
```

Issue Count by Category

Issues by Category	
Unreleased Resource: Database	6
Poor Error Handling: Overly Broad Throws	4
Hardcoded Domain in HTML	3
System Information Leak: HTML Comment in JSP	3
Access Control: Azure Storage	1
Azure Resource Manager Bad Practices: Cross-Tenant Replication	1
Azure Resource Manager Misconfiguration: Insecure Transport	1
Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy	1
Azure Resource Manager Misconfiguration: Public Access Allowed	1
Cross-Site Request Forgery	1
Insecure Transport: Azure Storage	1
Insecure Transport: Database	1
Insecure Transport: Weak SSL Protocol	1
J2EE Bad Practices: getConnection()	1
J2EE Misconfiguration: Excessive Session Timeout	1
J2EE Misconfiguration: Missing Error Handling	1
Log Forging (debug)	1
Password Management	1
Password Management: Hardcoded Password	1
Password Management: Password in Comment	1
Poor Logging Practice: Logger Not Declared Static Final	1
SQL Injection	1
System Information Leak: Internal	1

Issue Breakdown by Analysis

Issues by Analysis