OWASP Top 10 ... 2021

Apr 12, 2023

CoEAS Bhubaneswar

Executive Summary

Issues Overview

On Apr 12, 2023, a source code review was performed over the ndcbbsrweb code base. 390 files, 2,377 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 87 reviewed findings were uncovered during the analysis.

Issues by OWASP Top 10 2021	
A08 Software and Data Integrity Failures	9
A05 Security Misconfiguration	17
A04 Insecure Design	8
A03 Injection	7
A02 Cryptographic Failures	2
A01 Broken Access Control	12
<none>	32

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: D:/SCA/Year_2023/NDCBBSR/ndcbbsrweb_5th_Level/ndcbbsrweb
Number of Files: 390
Lines of Code: 2377
Build Label: <No Build Label>

Scan Information

Scan time: 00:57
SCA Engine version: 20.2.2.0003
Machine Name: DESKTOP-P8I04US
Username running scan: sanjukta

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:
com.example.ndcbbsrweb.NdcbbsrwebApplication.main

Environment Variables:
java.lang.System.getenv

File System:
org.apache.commons.io.FileUtils.readFileToByteArray

Private Information:
null.null.null
null.null.null
com.example.ndcbbsrweb.util.AesCrypto.decrypt
java.lang.System.getenv
javax.crypto.KeyGenerator.generateKey
javax.crypto.SecretKeyFactory.generateSecret

Java Properties:
java.lang.System.getProperty

System Information:
null.null.null
null.null.resolve
com.amazonaws.services.s3.AmazonS3.putObject
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.Throwable.getMessage

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:
(Disabled) If impact is not in range [2.5, 5.0] Then hide issue
(Disabled) If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

Audit guide not enabled

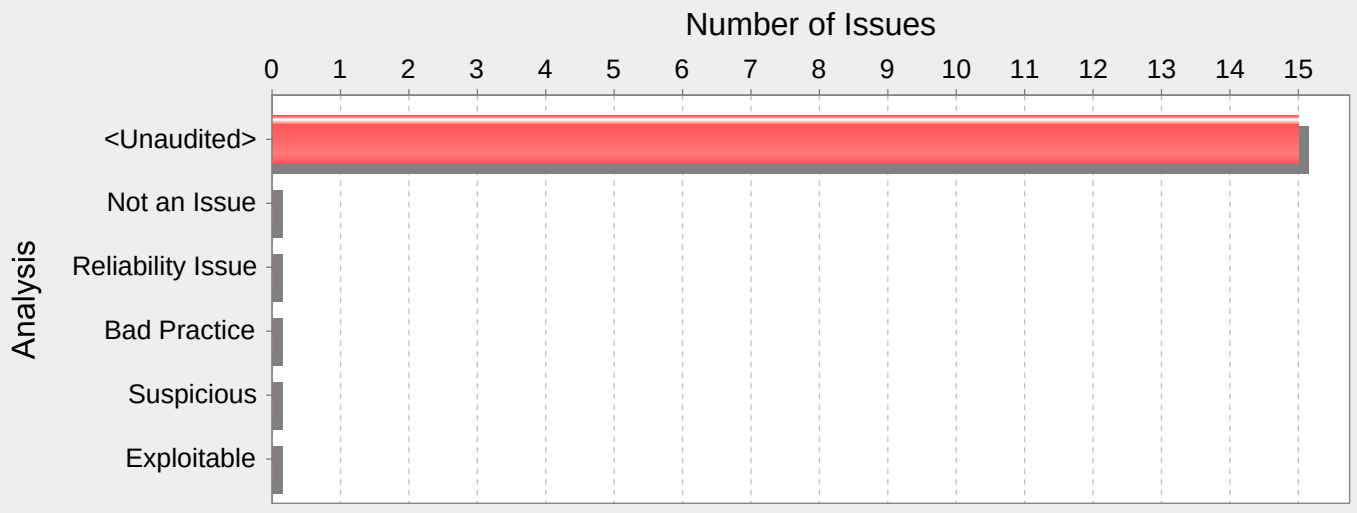
Results Outline

Overall number of results

The scan found 87 issues.

Vulnerability Examples by Category

Category: Poor Error Handling: Overly Broad Catch (15 Issues)



Abstract:

The catch block at GlobalDefaultExceptionHandler.java line 24 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.

Explanation:

Multiple catch blocks can get repetitive, but "condensing" catch blocks by catching a high-level class such as Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention.

Example: The following code excerpt handles three types of exceptions in an identical fashion.

```
try {
doExchange();
}
catch (IOException e) {
logger.error("doExchange failed", e);
}
catch (InvocationTargetException e) {
logger.error("doExchange failed", e);
}
catch (SQLException e) {
logger.error("doExchange failed", e);
}
```

At first blush, it may seem preferable to deal with these exceptions in a single catch block, as follows:

```
try {
doExchange();
}
catch (Exception e) {
logger.error("doExchange failed", e);
}
```

However, if doExchange() is modified to throw a new type of exception that should be handled in some different kind of way, the broad catch block will prevent the compiler from pointing out the situation. Further, the new catch block will now also handle exceptions derived from RuntimeException such as ClassCastException, and NullPointerException, which is not the programmer's intent.

Recommendations:

Do not catch broad exception classes such as Exception, Throwable, Error, or RuntimeException except at the very top level of the program or thread.

Tips:

1. The Fortify Secure Coding Rulepacks will not flag an overly broad catch block if the catch block in question immediately throws a new exception.

SecSecurityConfig.java, line 30 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at SecSecurityConfig.java line 30 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	SecSecurityConfig.java:30 CatchBlock()		

```
28                .addHeaderWriter(new StaticHeadersWriter("X-Content-Security-Policy", "script-src 'self'")).and()
29                .formLogin().loginPage("/parichayclient/dashboard").and().logout().permitAll();
30              } catch (Exception e) {
31               LOGGER.debug("configuration issue");
32              }
```

ObjStoreConfig.java, line 84 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at ObjStoreConfig.java line 84 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	ObjStoreConfig.java:84 CatchBlock()		

```
82              }
83
84              catch (Exception e) {
85               LOGGER.debug("object cannot be transferred");
86              }
```

ObjStoreConfig.java, line 65 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at ObjStoreConfig.java line 65 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	ObjStoreConfig.java:65 CatchBlock()		

```
63              }
64
65              catch (Exception ex) {
66               LOGGER.debug("object cannot be converted to data bytes");
67              }
```

AesCrypto.java, line 77 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at AesCrypto.java line 77 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	AesCrypto.java:77 CatchBlock()		

```
75              decryptCipher.init(Cipher.DECRYPT_MODE, key, spec);
```

```
76                    return decryptCipher.doFinal(encrypted);
77                   } catch (Exception e) {
78                    LOGGER.debug("AesCrypto.decrypt error");
79                    return null;
```

NdcServiceImpl.java, line 337 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at NdcServiceImpl.java line 337 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: NdcServiceImpl.java:337 CatchBlock()

```
335                   try {
336                    portalList = portals.findAll();
337                   } catch (Exception e) {
338                    LOGGER.debug("getting portal from db exception");
339                   }
```

UtkalUtil.java, line 74 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at UtkalUtil.java line 74 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: UtkalUtil.java:74 CatchBlock()

```
72                    try {
73                     no = Double.parseDouble(str);
74                    } catch (Exception e) {
75                    }
76                    return no;
```

ObjStoreConfig.java, line 44 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at ObjStoreConfig.java line 44 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: ObjStoreConfig.java:44 CatchBlock()

```
42                        .withPathStyleAccessEnabled(true).withClientConfiguration(clientConfiguration)
43                        .withCredentials(new AWSStaticCredentialsProvider(credentials)).build();
44                    } catch (Exception e) {
45                     LOGGER.debug("S3Client creation error");
46                    }
```

MailAuthSMTP.java, line 48 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at MailAuthSMTP.java line 48 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: MailAuthSMTP.java:48 CatchBlock()

```
46                    transport.sendMessage(message, message.getRecipients(Message.RecipientType.TO));
47                    transport.close();
48                   } catch (Exception e) {
49                    // TODO Auto-generated catch block
50                    LOGGER.debug("ERROR in sending mail : ", e.getMessage());
```

AESEncryption.java, line 115 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at AESEncryption.java line 115 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: AESEncryption.java:115 CatchBlock()

```
113
114                 decryptedText = decryptText(cipherTextFromHex, secretKeyConvertedFromStringKey);
115               } catch (Exception ex) {
116                 LOGGER.debug("Exception caught during decrypt the text string and get message : " +
ex.getMessage());
117               }
```

UtkalUtil.java, line 184 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at UtkalUtil.java line 184 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: UtkalUtil.java:184 CatchBlock()

```
182                 return true;
183               }
184               } catch (Exception ex) {
185                 LOGGER.debug("parsing error at isSafeLastLogin");
186                 return false;
```

TransactionFilter.java, line 37 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at TransactionFilter.java line 37 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: TransactionFilter.java:37 CatchBlock()

```
35                 try {
36                  tokenCheck = objHome.isTokenValid(req, res);
37                 } catch (Exception e) {
38                  LOGGER.debug("filter error");
39                 }
```

AESEncryption.java, line 99 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at AESEncryption.java line 99 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: AESEncryption.java:99 CatchBlock()

```
97                try {
98                 cipherText = encryptText(plainText, secretKeyConvertedFromStringKey);
99                } catch (Exception e) {
100                LOGGER.debug("AESENCRYPTION.encryptIntoHex encryptText Exception");
101               }
```

CheckImage.java, line 27 (Poor Error Handling: Overly Broad Catch)

Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at CheckImage.java line 27 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink:	CheckImage.java:27 CatchBlock()

```
25                  }
26
27                } catch (Exception e) {
28                 LOGGER.debug("image io error at isImage function");
29                }
```

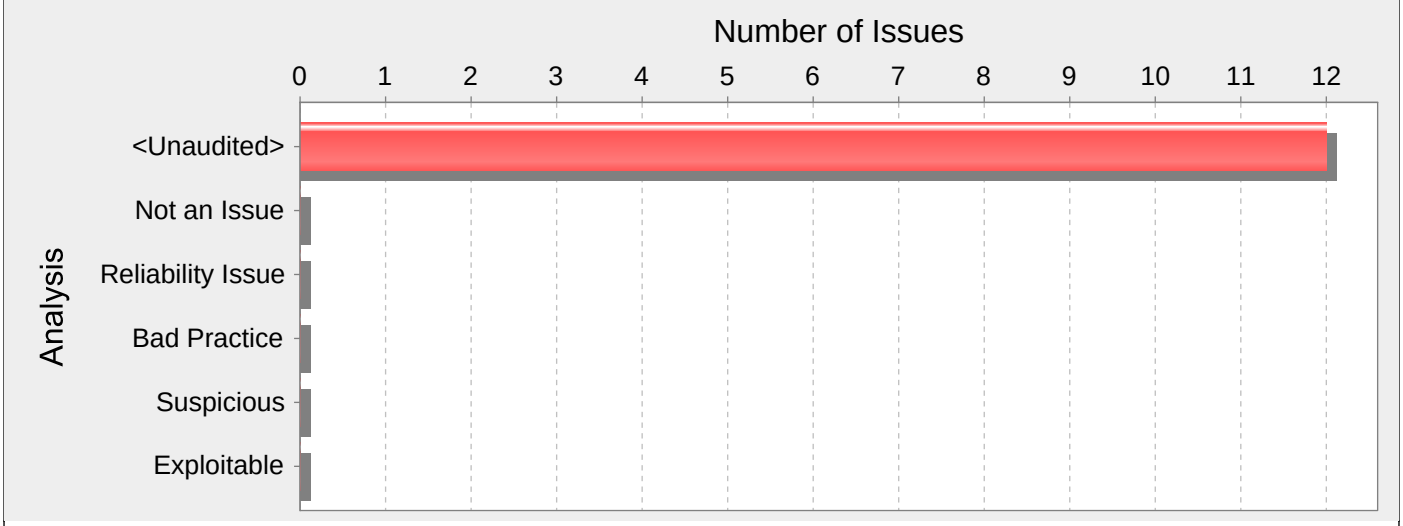
GlobalDefaultExceptionHandler.java, line 24 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at GlobalDefaultExceptionHandler.java line 24 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	GlobalDefaultExceptionHandler.java:24 CatchBlock()		

```
22                  try {
23                   throw e;
24                  } catch (Exception e1) {
25                   LOGGER.debug("error throw error");
26                  }
```

UtkalUtil.java, line 61 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at UtkalUtil.java line 61 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	UtkalUtil.java:61 CatchBlock()		

```
59                try {
60                 no = Integer.parseInt(str);
61                } catch (Exception e) {
62                }
63                return no;
```

Category: Path Manipulation (12 Issues)



Abstract:

Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, 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 file system.

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

For example, the program might 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 could provide a file name such as "../../tomcat/conf/server.xml", which causes the application to delete one of its own configuration files.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

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 adequate 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.

```
fis = new FileInputStream(cfg.getProperty("sub")+".txt");
amt = fis.read(arr);
out.println(arr);
```

Some think that in the mobile environment, classic vulnerabilities, such as path manipulation, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 3: The following code adapts Example 1 to the Android platform.

```
...
String rName = this.getIntent().getExtras().getString("reportName");
File rFile = getBaseContext().getFileStreamPath(rName);
...
rFile.delete();
...
```

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate values from which the user must select. With this approach, the user-provided input 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 maintain. Programmers often resort to implementing a deny list in these situations. A deny list is used to selectively reject or escape 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 list of characters that are permitted to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program performs custom input validation to your satisfaction, use the Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Implementation of an effective deny list is notoriously difficult. One should be skeptical if validation logic requires implementing a deny list. 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 deny list can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the 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.

AdminPanelController.java, line 1249 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, which allows them to access or modify otherwise protected files.		

Source: AdminPanelController.java:796 saveNews(0)

```
794
795     @PostMapping("/saveNews")
796     public String saveNews(@RequestParam("image") MultipartFile file, LatestNewsModal
        newsmodal,
797       HttpServletRequest request) {
```

Sink: AdminPanelController.java:1249 java.io.File.File()

```
1247      Date d = new Date();
1248      String filename = d.getTime() + "." + extension;
1249      File convFile = new File(filename);
1250      FileOutputStream fos = null;
1251      try {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.		

Source: ObjStoreConfig.java:28 java.lang.System.getenv()

```
26    private static final String accessKey = System.getenv("accessKey");// ObjectStore
      accessKey
27    private static final String secretKey = System.getenv("secretKey");// ObjectStore
      secretKey
28    private static final String bucketName = System.getenv("bucketName");// BucketName
29    private static final String endPoint = System.getenv("staasEndPoint");
30    private static final Logger LOGGER = LogManager.getLogger(AdminPanelController.class);
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75      AmazonS3 s3C = getS3Client();
76      s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77      return 1;
78     } catch (AmazonServiceException ase) {
```

AdminPanelController.java, line 1249 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical

Kingdom:	Input Validation and Representation
Abstract:	Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, which allows them to access or modify otherwise protected files.
Source:	AdminPanelController.java:487 saveBanner(1)

```
485
486    @PostMapping("/saveBanner")
487    public String saveBanner(BannerModal banner, @RequestParam("image") MultipartFile
file,
488        HttpServletRequest request) {
```

Sink:	AdminPanelController.java:1249 java.io.File.File()

```
1247     Date d = new Date();
1248     String filename = d.getTime() + "." + extension;
1249     File convFile = new File(filename);
1250     FileOutputStream fos = null;
1251     try {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	Critical	Folder	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.		
Source:	AdminPanelController.java:829 updateNews(0)		

```
827
828    @PostMapping("/updateNews")
829    public String updateNews(@RequestParam(value = "file", required = false) MultipartFile
file,
830        LatestNewsModal newsmodal, HttpServletRequest request) {
```

Sink:	ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75       AmazonS3 s3C = getS3Client();
76       s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77       return 1;
78      } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	Critical	Folder	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.		
Source:	AdminPanelController.java:574 saveGallery(1)		

```
572
573    @PostMapping("/saveGallery")
574    public String saveGallery(BannerModal banner, @RequestParam("image") MultipartFile
file,
575        HttpServletRequest request) {
```

Sink:	ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75       AmazonS3 s3C = getS3Client();
76       s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77       return 1;
78      } catch (AmazonServiceException ase) {
```

AdminPanelController.java, line 1249 (Path Manipulation)

Fortify Priority:	Critical	Folder	Critical
Kingdom:	Input Validation and Representation		

Abstract:	Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, which allows them to access or modify otherwise protected files.

Source: AdminPanelController.java:829 updateNews(0)

```
827
828     @PostMapping("/updateNews")
829     public String updateNews(@RequestParam(value = "file", required = false) MultipartFile
file,
830       LatestNewsModal newsmodal, HttpServletRequest request) {
```

Sink: AdminPanelController.java:1249 java.io.File.File()

```
1247      Date d = new Date();
1248      String filename = d.getTime() + "." + extension;
1249      File convFile = new File(filename);
1250      FileOutputStream fos = null;
1251      try {
```

AdminPanelController.java, line 1249 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		

Abstract:	Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, which allows them to access or modify otherwise protected files.

Source: AdminPanelController.java:574 saveGallery(1)

```
572
573     @PostMapping("/saveGallery")
574     public String saveGallery(BannerModal banner, @RequestParam("image") MultipartFile
file,
575       HttpServletRequest request) {
```

Sink: AdminPanelController.java:1249 java.io.File.File()

```
1247      Date d = new Date();
1248      String filename = d.getTime() + "." + extension;
1249      File convFile = new File(filename);
1250      FileOutputStream fos = null;
1251      try {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		

Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.

Source: AdminPanelController.java:920 saveHighlight(0)

```
918
919     @PostMapping("/saveHighlight")
920     public String saveHighlight(@RequestParam("image") MultipartFile file, boolean
isImage, String content,
921       HttpServletRequest request) {
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75        AmazonS3 s3C = getS3Client();
76        s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77        return 1;
78      } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		

Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.

Source: AdminPanelController.java:796 saveNews(0)

```
794
795   @PostMapping("/saveNews")
796   public String saveNews(@RequestParam("image") MultipartFile file, LatestNewsModal
      newsmodal,
797     HttpServletRequest request) {
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75      AmazonS3 s3C = getS3Client();
76      s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77      return 1;
78    } catch (AmazonServiceException ase) {
```

AdminPanelController.java, line 1249 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to File() at AdminPanelController.java line 1249, which allows them to access or modify otherwise protected files.		

Source: AdminPanelController.java:920 saveHighlight(0)

```
918
919   @PostMapping("/saveHighlight")
920   public String saveHighlight(@RequestParam("image") MultipartFile file, boolean
      isImage, String content,
921     HttpServletRequest request) {
```

Sink: AdminPanelController.java:1249 java.io.File.File()

```
1247    Date d = new Date();
1248    String filename = d.getTime() + "." + extension;
1249    File convFile = new File(filename);
1250    FileOutputStream fos = null;
1251    try {
```

ObjStoreConfig.java, line 76 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to PutObjectRequest() at ObjStoreConfig.java line 76, which allows them to access or modify otherwise protected files.		

Source: AdminPanelController.java:487 saveBanner(1)

```
485
486   @PostMapping("/saveBanner")
487   public String saveBanner(BannerModal banner, @RequestParam("image") MultipartFile
      file,
488     HttpServletRequest request) {
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75      AmazonS3 s3C = getS3Client();
76      s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77      return 1;
78    } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 54 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		

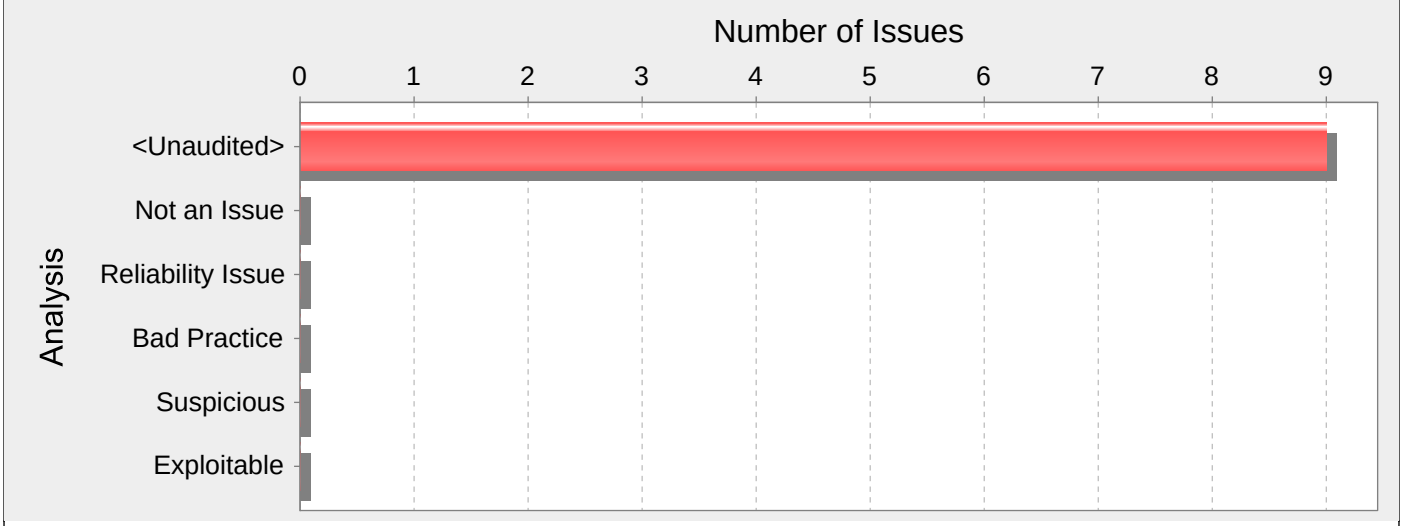
Abstract:	Attackers can control the file system path argument to File() at ObjStoreConfig.java line 54, which allows them to access or modify otherwise protected files.
Source:	HomeController.java:117 getResources(0)

```
115
116                 @RequestMapping(value = { "/getdata" }, method = RequestMethod.GET)
117                 public ModelAndView getResources(@RequestParam("rid") String rid, @RequestParam("dir")
                    String dir,
118                    HttpServletResponse response, Model model) {
119                   rid = UtkalUtil.safeLogMsg(100, rid); // sanitized rid
```

Sink:	ObjStoreConfig.java:54 java.io.File.File()

```
52                  //   AmazonS3 s3C = getS3Client();
53                  //   GetObjectRequest objectRequest = new GetObjectRequest(bucketName, keyName);
54                     File tempFile = new File(keyName);
55                  //   ObjectMetadata sobj = s3C.getObject(objectRequest, tempFile);
56                     byte[] data = FileUtils.readFileToByteArray(tempFile);
```

Category: Mass Assignment: Insecure Binder Configuration (9 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: Using Spring MVC with no additional configuration, the following controller method will bind the HTTP request parameters to any attribute in the User or Details classes:

```
@RequestMapping(method = RequestMethod.POST)
public String registerUser(@ModelAttribute("user") User user, BindingResult result, SessionStatus status) {
if (db.save(user).hasErrors()) {
return "CustomerForm";
} else {
status.setComplete();
return "CustomerSuccess";
}
}
```

Where User class is defined as:

```
public class User {
private String name;
private String lastname;
private int age;
private Details details;
// Public Getters and Setters
...
}
```

and Details class is defined as:

```
public class Details {
private boolean is_admin;
private int id;
private Date login_date;
// Public Getters and Setters
...
```

```
}
```

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 figure out 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:

Spring MVC:

It is possible to control which HTTP request parameters will be used in the binding process and which ones will be ignored.

In Spring MVC applications using @ModelAttribute annotated parameters, the binder can be configured to control which attributes should be bound. In order to do so, a method can be annotated with @InitBinder so that the framework will inject a reference to the Spring Model Binder. The Spring Model Binder can be configured to control the attribute binding process with the setAllowedFields and setDisallowedFields methods. Spring MVC applications extending BaseCommandController can override the initBinder(HttpServletRequest request, ServletRequestDataBinder binder) method in order to get a reference to the Spring Model Binder.

Example 2: The Spring Model Binder (3.x) is configured to disallow the binding of sensitive attributes:

```
final String[] DISALLOWED_FIELDS = new String[]{"details.role", "details.age", "is_admin"};
@InitBinder
public void initBinder(WebDataBinder binder) {
binder.setDisallowedFields(DISALLOWED_FIELDS);
}
```

Example 3: The Spring Model Binder (2.x) is configured to disallow the binding of sensitive attributes:

```
@Override
protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception {
binder.setDisallowedFields(new String[]{"details.role", "details.age", "is_admin"});
}
```

In Spring MVC Applications using @RequestBody annotated parameters, the binding process is handled by HttpMessageConverter instances which will use libraries such as Jackson and JAXB to convert the HTTP request body into Java Objects. These libraries offer annotations to control which fields should be allowed or disallowed. For example, for the Jackson JSON library, the @JsonIgnore annotation can be used to prevent a field from being bound to the request.

Example 4: A controller method binds an HTTP request to an instance of the Employee class using the @RequestBody annotation.

```
@RequestMapping(value="/add/employee", method=RequestMethod.POST, consumes="text/html")
public void addEmployee(@RequestBody Employee employee){
// Do something with the employee object.
}
```

The application uses the default Jackson HttpMessageConverter to bind JSON HTTP requests to the Employee class. In order to prevent the binding of the is_admin sensitive field, use the @JsonIgnore annotation:

```
public class Employee {
@JsonIgnore
private boolean is_admin;
...
// Public Getters and Setters
...
}
```

Note: Check the following REST frameworks information for more details on how to configure Jackson and JAXB annotations.

Apache Struts:

Struts 1 and 2 will only bind HTTP request parameters to those Actions or ActionForms attributes which have an associated public setter accessor. If an attribute should not be bound to the request, its setter should be made private.

Example 5: Configure a private setter so that Struts framework will not automatically bind any HTTP request parameter:

```
private String role;
```

```
private void setRole(String role)  {
this.role = role;
}
```

REST frameworks:

Most REST frameworks will automatically bind any HTTP request bodies with content type JSON or XML to a model object. Depending on the libraries used for JSON and XML processing, there will be different ways of controlling the binding process. The following are some examples for JAXB (XML) and Jackson (JSON):

Example 6: Models bound from XML documents using Oracle's JAXB library can control the binding process using different annotations such as @XmlAccessorType, @XmlAttribute, @XmlElement and @XmlTransient. The binder can be told not to bind any attributes by default, by annotating the models using the @XmlAccessorType annotation with the value XmlAccessType.NONE and then selecting which fields should be bound using @XmlAttribute and @XmlElement annotations:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class User {
private String role;
private String name;
@XmlAttribute
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getRole() {
return role;
}
public void setRole(String role) {
this.role = role;
}
```

Example 7: Models bound from JSON documents using the Jackson library can control the binding process using different annotations such as @JsonIgnore, @JsonIgnoreProperties, @JsonIgnoreType and @JsonInclude. The binder can be told to ignore certain attributes by annotating them with @JsonIgnore annotation:

```
public class User {
@JsonIgnore
private String role;
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getRole() {
return role;
}
public void setRole(String role) {
this.role = role;
}
```

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.

AdminPanelController.java, line 346 (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:	AdminPanelController.java:346 Function: saveTeam()		

```
344
345     @PostMapping("/saveTeam")
346     public String saveTeam(@ModelAttribute("team") teamModal team) {
347
348       if (PriviledgeCheckAdmin() == false) {
```

AdminPanelController.java, line 796 (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:	AdminPanelController.java:796 Function: saveNews()		

```
794
795     @PostMapping("/saveNews")
796     public String saveNews(@RequestParam("image") MultipartFile file, LatestNewsModal
        newsmodal,
797       HttpServletRequest request) {
```

AdminPanelController.java, line 387 (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:	AdminPanelController.java:387 Function: updateTeam()		

```
385
386     @PostMapping("/updateTeam")
387     public String updateTeam(@ModelAttribute("team") teamModal team) {
388
389       if (PriviledgeCheckAdmin() == false)
```

AdminPanelController.java, line 574 (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:	AdminPanelController.java:574 Function: saveGallery()		

```
572
573     @PostMapping("/saveGallery")
574     public String saveGallery(BannerModal banner, @RequestParam("image") MultipartFile
        file,
575       HttpServletRequest request) {
```

AdminPanelController.java, line 487 (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:	AdminPanelController.java:487 Function: saveBanner()

```
485
486                @PostMapping("/saveBanner")
487                public String saveBanner(BannerModal banner, @RequestParam("image") MultipartFile
file,
488                   HttpServletRequest request) {
```

AdminPanelController.java, line 829 (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:	AdminPanelController.java:829 Function: updateNews()		

```
827
828                @PostMapping("/updateNews")
829                public String updateNews(@RequestParam(value = "file", required = false) MultipartFile
file,
830                   LatestNewsModal newsmodal, HttpServletRequest request) {
```

AdminPanelController.java, line 1075 (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:	AdminPanelController.java:1075 Function: savePricing()		

```
1073
1074               @PostMapping("/savePricing")
1075               public String savePricing(@ModelAttribute("pricing") PricingModal pricing) {
1076
1077                 if (!IpLoginCheck()) {
```

AdminPanelController.java, line 1159 (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:	AdminPanelController.java:1159 Function: updateContact()		

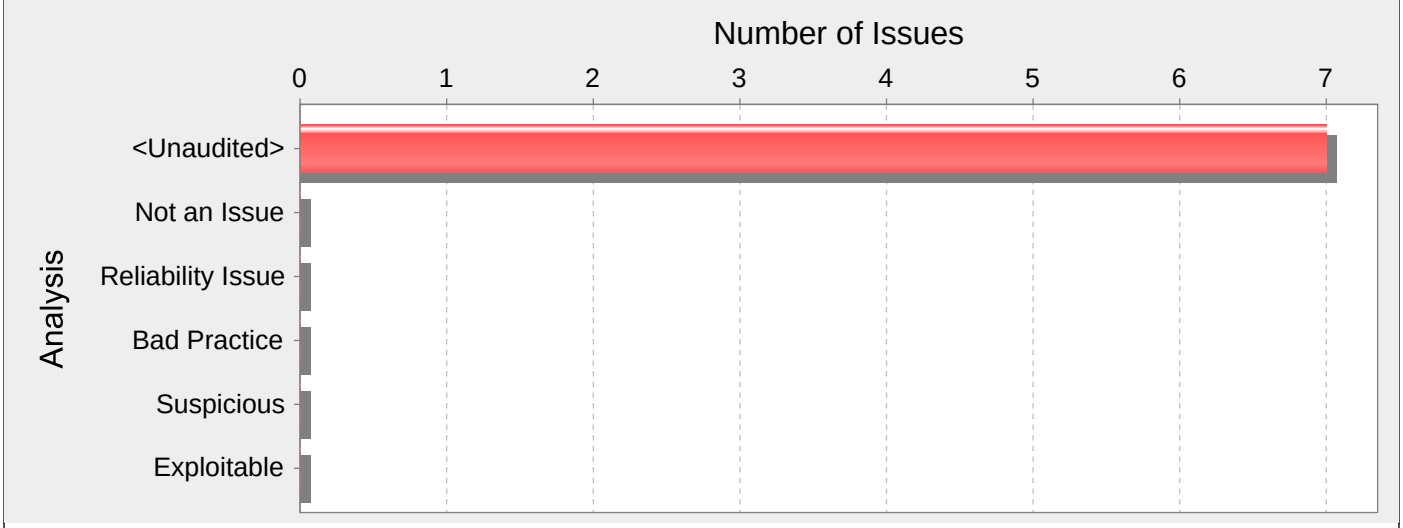
```
1157
1158               @PostMapping("/updateContact")
1159               public String updateContact(@ModelAttribute("contact") ContactUSModal contactus) {
1160
1161                 if (!IpLoginCheck()) {
```

AdminPanelController.java, line 661 (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:	AdminPanelController.java:661 Function: saveTender()		

```
659
660                @PostMapping("/saveTender")
661                public String saveTender(@ModelAttribute("tender") TenderAndNotificationModal tender)
{
662
663                 if (!IpLoginCheck()) {
```

Category: Cookie Security: Cookie not Sent Over SSL (7 Issues)



Abstract:

A cookie is created without the Secure flag set to true.

Explanation:

Modern web browsers support a Secure flag for each cookie. If the flag is set, the browser will only send the cookie over HTTPS. Sending cookies over an unencrypted channel can expose them to network sniffing attacks, so the secure flag helps keep a cookie's value confidential. This is especially important if the cookie contains private data or carries a session identifier.

Example 1: In the following example, a cookie is added to the response without setting the Secure flag.

```
Cookie cookie = new Cookie("emailCookie", email);
response.addCookie(cookie);
```

If your application uses both HTTPS and HTTP but does not set the Secure flag, cookies sent during an HTTPS request will also be sent during subsequent HTTP requests. Sniffing network traffic over unencrypted wireless connections is a trivial task for attackers, so sending cookies (especially those with session IDs) over HTTP can result in application compromise.

Recommendations:

Set the Secure flag on all new cookies in order to instruct browsers not to send these cookies in the clear. Do this by calling setSecure(true).

Example 2:

```
Cookie cookie = new Cookie("emailCookie", email);
cookie.setSecure(true);
response.addCookie(cookie);
```

AdminPanelController.java, line 241 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:241 addCookie(cookie2)		

```
239
240         response.addCookie(cookie1);
241         response.addCookie(cookie2);
242         response.addCookie(cookie3);
243         response.addCookie(cookie4);
```

AdminPanelController.java, line 243 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:243 addCookie(cookie4)		

```
241         response.addCookie(cookie2);
242         response.addCookie(cookie3);
243         response.addCookie(cookie4);
```

```
244                    } catch (Exception e) {
245                     LOGGER.debug("AdminPanel.homepage Exception");
```

AdminPanelController.java, line 218 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:218 cookie2 = new Cookie(...)		

```
216                    Cookie cookie1 = new Cookie("un", URLEncoder.encode(
217                      aesCrypto.encrypt(seedKey, (String) session.getAttribute("userName")),
                    StandardCharsets.UTF_8));
218                    Cookie cookie2 = new Cookie("si",
219                      URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
                    session.getAttribute("sessionId")),
220                        StandardCharsets.UTF_8));
```

AdminPanelController.java, line 242 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:242 addCookie(cookie3)		

```
240                    response.addCookie(cookie1);
241                    response.addCookie(cookie2);
242                    response.addCookie(cookie3);
243                    response.addCookie(cookie4);
244                   } catch (Exception e) {
```

AdminPanelController.java, line 216 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:216 cookie1 = new Cookie(...)		

```
214
215                   try {
216                    Cookie cookie1 = new Cookie("un", URLEncoder.encode(
217                      aesCrypto.encrypt(seedKey, (String) session.getAttribute("userName")),
                    StandardCharsets.UTF_8));
218                    Cookie cookie2 = new Cookie("si",
```

AdminPanelController.java, line 221 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:221 cookie3 = new Cookie(...)		

```
219                      URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
                    session.getAttribute("sessionId")),
220                        StandardCharsets.UTF_8));
221                    Cookie cookie3 = new Cookie("bi",
222                      URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
                    session.getAttribute("browserId")),
223                        StandardCharsets.UTF_8));
```

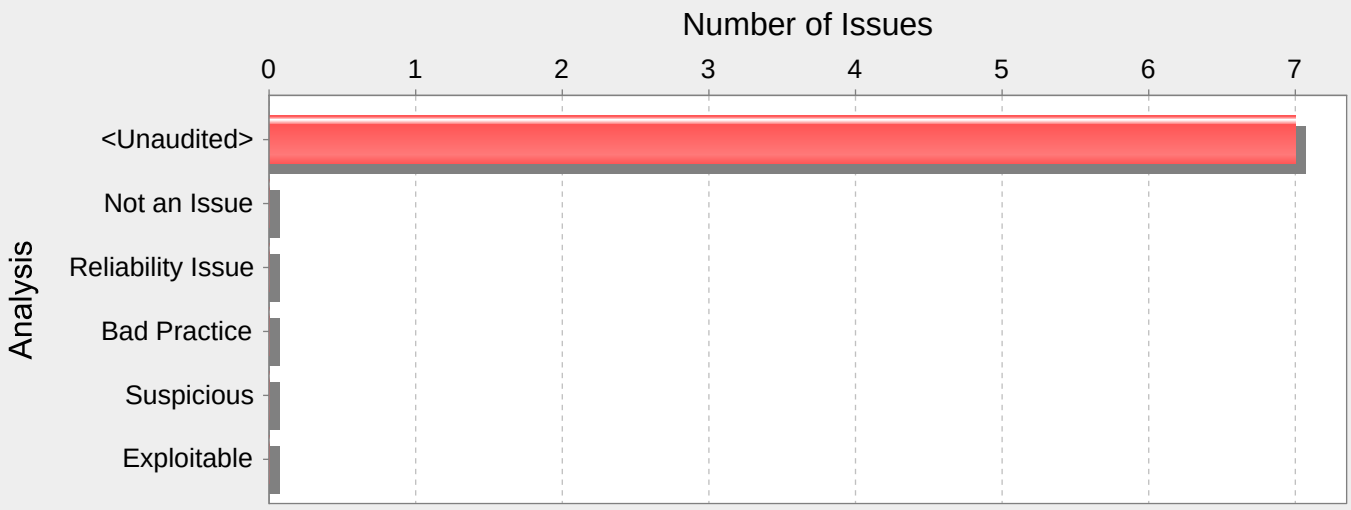
AdminPanelController.java, line 240 (Cookie Security: Cookie not Sent Over SSL)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	A cookie is created without the Secure flag set to true.		
Sink:	AdminPanelController.java:240 addCookie(cookie1)		

```
238                    cookie4.setDomain("ndcbbsr.nic.in");
239
```

```
240                         response.addCookie(cookie1);
241                         response.addCookie(cookie2);
242                         response.addCookie(cookie3);
```

Category: Poor Error Handling: Overly Broad Throws (7 Issues)



Abstract:

The method home() in AdminPanelController.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.

Explanation:

Declaring a method to throw Exception or Throwable makes it difficult for callers to do good error handling and error recovery. Java's exception mechanism is set up to make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.

Example: The following method throws three types of exceptions.

```
public void doExchange()
throws IOException, InvocationTargetException,
SQLException {
...
}
```

While it might seem tidier to write

```
public void doExchange()
throws Exception {
...
}
```

doing so hampers the caller's ability to understand and handle the exceptions that occur. Further, if a later revision of doExchange() introduces a new type of exception that should be treated differently than previous exceptions, there is no easy way to enforce this requirement.

Recommendations:

Do not declare methods to throw Exception or Throwable. If the exceptions thrown by a method are not recoverable or should not generally be caught by the caller, consider throwing unchecked exceptions rather than checked exceptions. This can be accomplished by implementing exception classes that extend RuntimeException or Error instead of Exception, or add a try/catch wrapper in your method to convert checked exceptions to unchecked exceptions.

AESEncryption.java, line 106 (Poor Error Handling: Overly Broad Throws)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method decyText() in AESEncryption.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		

Sink: AESEncryption.java:106 Function: decyText()

```
104  }
105
106  public static String decyText(String encrytext) throws Exception {
107   String decryptedText = null;
108   try {
```

AesCrypto.java, line 41 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method encrypt() in AesCrypto.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	AesCrypto.java:41 Function: encrypt()		

```
39  private final SecureRandom random = new SecureRandom();
40
41  public String encrypt(String keyString, String plaintext) throws Exception {
42   byte[] iv = new byte[IV_LENGTH_BYTE];
43   random.nextBytes(iv);
```

AdminPanelController.java, line 1282 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method home() in AdminPanelController.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	AdminPanelController.java:1282 Function: home()		

```
1280
1281  @GetMapping("/dashboard")
1282  protected String home(HttpServletRequest request, HttpServletResponse response)
1283    throws InterruptedException, IOException, Throwable {
```

AesCrypto.java, line 61 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method decrypt() in AesCrypto.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	AesCrypto.java:61 Function: decrypt()		

```
59  }
60
61  public byte[] decrypt(String encMsg, String keyString) throws Exception {
62   try {
63    byte[] cipherMessage = Base64.getDecoder().decode(encMsg.getBytes());
```

MailAuthSMTP.java, line 27 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method SendMail() in MailAuthSMTP.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	MailAuthSMTP.java:27 Function: SendMail()		

```
25  private static final Logger LOGGER = LogManager.getLogger(MailAuthSMTP.class);
26
27  public void SendMail(String email, String msg, String subject) throws Exception {
28   Properties props = new Properties();
29   props.put("mail.transport.protocol", "smtp");
```

AESEncryption.java, line 32 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method encryptText() in AESEncryption.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	AESEncryption.java:32 Function: encryptText()		

```
30  }
31
32  public static byte[] encryptText(String plainText, SecretKey secKey) throws Exception
{
```

```
33                  Cipher aesCipher = Cipher.getInstance("AES/CCM/NoPadding", "BC");
34                  aesCipher.init(Cipher.ENCRYPT_MODE, secKey);
```

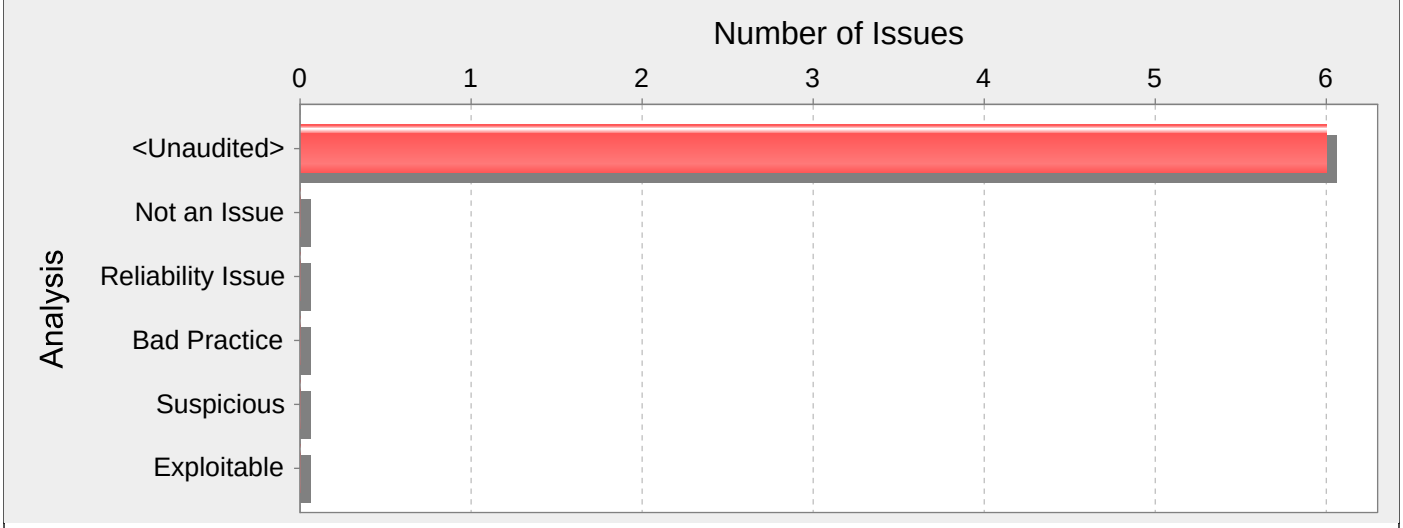
AESEncryption.java, line 25 (Poor Error Handling: Overly Broad Throws)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getSecretEncryptionKey() in AESEncryption.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		

Sink: AESEncryption.java:25 Function: getSecretEncryptionKey()

```
23                  private static Logger LOGGER = LogManager.getLogger(AESEncryption.class);
24
25                  public static SecretKey getSecretEncryptionKey() throws Exception {
26                  KeyGenerator generator = KeyGenerator.getInstance("AES");
27                  generator.init(128);
```

Category: Often Misused: File Upload (6 Issues)



Abstract:

A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 487 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.

Explanation:

Regardless of the language a program is written in, the most devastating attacks often involve remote code execution, whereby an attacker succeeds in executing malicious code in the program's context. If attackers are allowed to upload files to a directory that is accessible from the Web and cause these files to be passed to a code interpreter (e.g. JSP/ASPX/PHP), then they can cause malicious code contained in these files to execute on the server.

Example: The following Spring MVC controller class has a parameter than can be used to handle uploaded files.

```
@Controller
public class MyFormController {
...
@RequestMapping("/test")
public String uploadFile (org.springframework.web.multipart.MultipartFile file) {
...
}  ...
}
```

Even if a program stores uploaded files under a directory that isn't accessible from the Web, attackers might still be able to leverage the ability to introduce malicious content into the server environment to mount other attacks. If the program is susceptible to path manipulation, command injection, or dangerous file inclusion vulnerabilities, then an attacker might upload a file with malicious content and cause the program to read or execute it by exploiting another vulnerability.

Recommendations:

Do not accept attachments if they can be avoided. If a program must accept attachments, then restrict the ability of an attacker to supply malicious content by only accepting the specific types of content the program expects. Most attacks that rely on uploaded content require that attackers be able to supply content of their choosing. Placing restrictions on the content the program will accept will greatly limit the range of possible attacks. Check file names, extensions, and file content to make sure they are all expected and acceptable for use by the application. Make it difficult for the attacker to determine the name and location of uploaded files. Such solutions are often program-specific and vary from storing uploaded files in a directory with a name generated from a strong random value when the program is initialized to assigning each uploaded file a random name and tracking them with entries in a database.

AdminPanelController.java, line 487 (Often Misused: File Upload)

Fortify Priority:	Medium	**Folder**	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 487 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:487 Function: saveBanner()		

```
485
486     @PostMapping("/saveBanner")
```

```
487                public String saveBanner(BannerModal banner, @RequestParam("image") MultipartFile
file,
488                   HttpServletRequest request) {
```

AdminPanelController.java, line 574 (Often Misused: File Upload)			
Fortify Priority:	Medium	Folder	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 574 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:574 Function: saveGallery()		

```
572
573                @PostMapping("/saveGallery")
574                public String saveGallery(BannerModal banner, @RequestParam("image") MultipartFile
file,
575                   HttpServletRequest request) {
```

AdminPanelController.java, line 829 (Often Misused: File Upload)			
Fortify Priority:	Medium	Folder	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 829 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:829 Function: updateNews()		

```
827
828                @PostMapping("/updateNews")
829                public String updateNews(@RequestParam(value = "file", required = false) MultipartFile
file,
830                   LatestNewsModal newsmodal, HttpServletRequest request) {
```

AdminPanelController.java, line 1243 (Often Misused: File Upload)			
Fortify Priority:	Medium	Folder	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 1243 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:1243 Function: convertMultiPartToFile()		

```
1241               *
1242
                   ****************************************************************************
                   ***********************/
1243
1244               private File convertMultiPartToFile(MultipartFile file) {
1245                String extension = UtkalUtil.replaceSpecialChars("([^A-Za-z])",
```

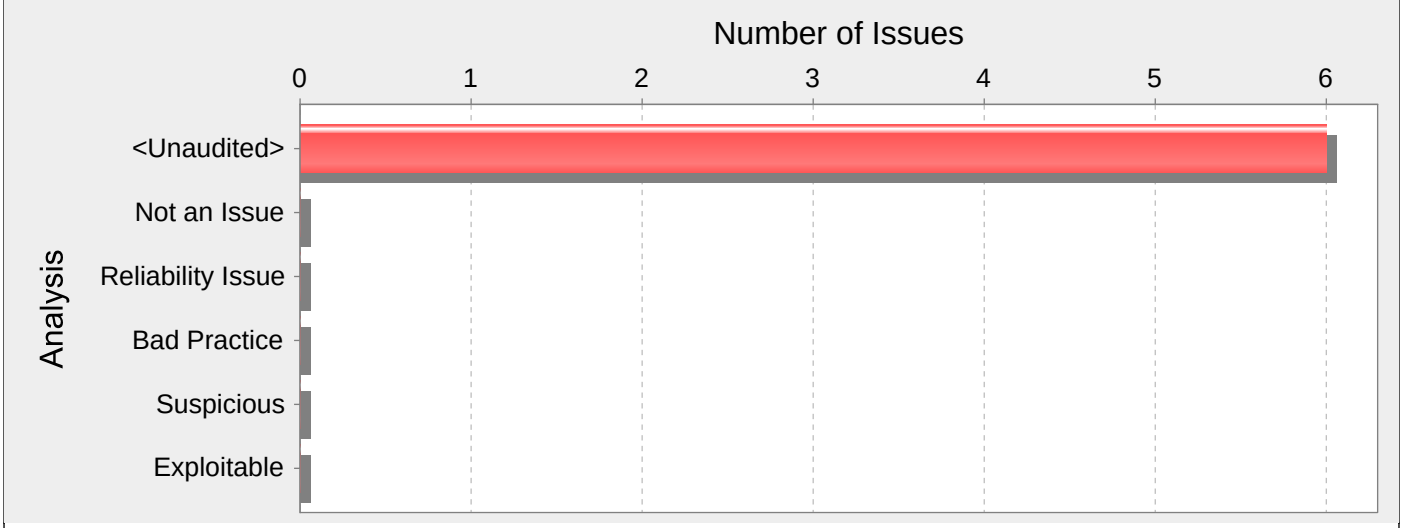
AdminPanelController.java, line 796 (Often Misused: File Upload)			
Fortify Priority:	Medium	Folder	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 796 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:796 Function: saveNews()		

```
794
795                @PostMapping("/saveNews")
796                public String saveNews(@RequestParam("image") MultipartFile file, LatestNewsModal
newsmodal,
797                   HttpServletRequest request) {
```

AdminPanelController.java, line 920 (Often Misused: File Upload)			
Fortify Priority:	Medium	**Folder**	Medium
Kingdom:	API Abuse		
Abstract:	A parameter of type org.springframework.web.multipart.MultipartFile in AdminPanelController.java on line 920 is used by the Spring MVC framework to set uploaded files. Permitting users to upload files can allow attackers to inject dangerous content or malicious code to run on the server.		
Sink:	AdminPanelController.java:920 Function: saveHighlight()		

```
918
919     @PostMapping("/saveHighlight")
920     public String saveHighlight(@RequestParam("image") MultipartFile file, boolean
        isImage, String content,
921       HttpServletRequest request) {
```

Category: Resource Injection (6 Issues)



Abstract:

Attackers are able to control the resource identifier argument to create() at AdminPanelController.java line 1335, which could enable them to access or modify otherwise protected system resources.

Explanation:

A resource injection issue occurs when the following two conditions are met:

1. An attacker is able to specify the identifier used to access a system resource.

For example, an attacker may be able to specify a port number to be used to connect to a network resource.

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 transmit sensitive information to a third-party server.

Note: Resource injections involving resources stored on the file system are reported in a separate category named path manipulation. See the path manipulation description for further details of this vulnerability.

Example 1: The following code uses a port number read from an HTTP request to create a socket.

```
String remotePort = request.getParameter("remotePort");
...
ServerSocket srvr = new ServerSocket(remotePort);
Socket skt = srvr.accept();
...
```

Some think that in the mobile world, classic web application vulnerabilities, such as resource injection, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 2: The following code uses a URL read from an Android intent to load the page in WebView.

```
...
WebView webview = new WebView(this);
setContentView(webview);
String url = this.getIntent().getExtras().getString("url");
webview.loadUrl(url);
...
```

The kind of resource affected by user input indicates the kind of content that may be dangerous. For example, data containing special characters like period, slash, and backslash are risky when used in methods that interact with the file system. Similarly, data that contains URLs and URIs is risky for functions that create remote connections.

Recommendations:

The best way to prevent resource injection 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 maintain. Programmers often resort to implementing a deny list in these situations. A deny list is used to selectively reject or escape 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 list of characters that are permitted to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program performs custom input validation to your satisfaction, use the Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Implementation of an effective deny list is notoriously difficult. One should be skeptical if validation logic requires implementing a deny list. 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 deny list can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the 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.

ObjStoreConfig.java, line 76 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	Attackers are able to control the resource identifier argument to PutObjectRequest() at ObjStoreConfig.java line 76, which could enable them to access or modify otherwise protected system resources.		

Source: AdminPanelController.java:487 saveBanner(1)

```
485
486    @PostMapping("/saveBanner")
487    public String saveBanner(BannerModal banner, @RequestParam("image") MultipartFile
       file,
488        HttpServletRequest request) {
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75        AmazonS3 s3C = getS3Client();
76        s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77        return 1;
78      } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	Attackers are able to control the resource identifier argument to PutObjectRequest() at ObjStoreConfig.java line 76, which could enable them to access or modify otherwise protected system resources.		

Source: AdminPanelController.java:574 saveGallery(1)

```
572
573    @PostMapping("/saveGallery")
574    public String saveGallery(BannerModal banner, @RequestParam("image") MultipartFile
       file,
575        HttpServletRequest request) {
```

Sink: ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75        AmazonS3 s3C = getS3Client();
76        s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77        return 1;
78      } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low

Kingdom:	Input Validation and Representation
Abstract:	Attackers are able to control the resource identifier argument to PutObjectRequest() at ObjStoreConfig.java line 76, which could enable them to access or modify otherwise protected system resources.
Source:	AdminPanelController.java:920 saveHighlight(0)

```
918
919                 @PostMapping("/saveHighlight")
920                 public String saveHighlight(@RequestParam("image") MultipartFile file, boolean
isImage, String content,
921                     HttpServletRequest request) {
```

Sink:	ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75                    AmazonS3 s3C = getS3Client();
76                    s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77                    return 1;
78                  } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	Attackers are able to control the resource identifier argument to PutObjectRequest() at ObjStoreConfig.java line 76, which could enable them to access or modify otherwise protected system resources.		
Source:	AdminPanelController.java:829 updateNews(0)		

```
827
828                 @PostMapping("/updateNews")
829                 public String updateNews(@RequestParam(value = "file", required = false) MultipartFile
file,
830                     LatestNewsModal newsmodal, HttpServletRequest request) {
```

Sink:	ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75                    AmazonS3 s3C = getS3Client();
76                    s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77                    return 1;
78                  } catch (AmazonServiceException ase) {
```

ObjStoreConfig.java, line 76 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	Attackers are able to control the resource identifier argument to PutObjectRequest() at ObjStoreConfig.java line 76, which could enable them to access or modify otherwise protected system resources.		
Source:	AdminPanelController.java:796 saveNews(0)		

```
794
795                 @PostMapping("/saveNews")
796                 public String saveNews(@RequestParam("image") MultipartFile file, LatestNewsModal
newsmodal,
797                     HttpServletRequest request) {
```

Sink:	ObjStoreConfig.java:76 com.amazonaws.services.s3.model.PutObjectRequest.PutObjectRequest()

```
74
75                    AmazonS3 s3C = getS3Client();
76                    s3C.putObject(new PutObjectRequest(bucketName, keyName, file));
77                    return 1;
78                  } catch (AmazonServiceException ase) {
```

AdminPanelController.java, line 1335 (Resource Injection)

Fortify Priority:	Low	**Folder**	Low

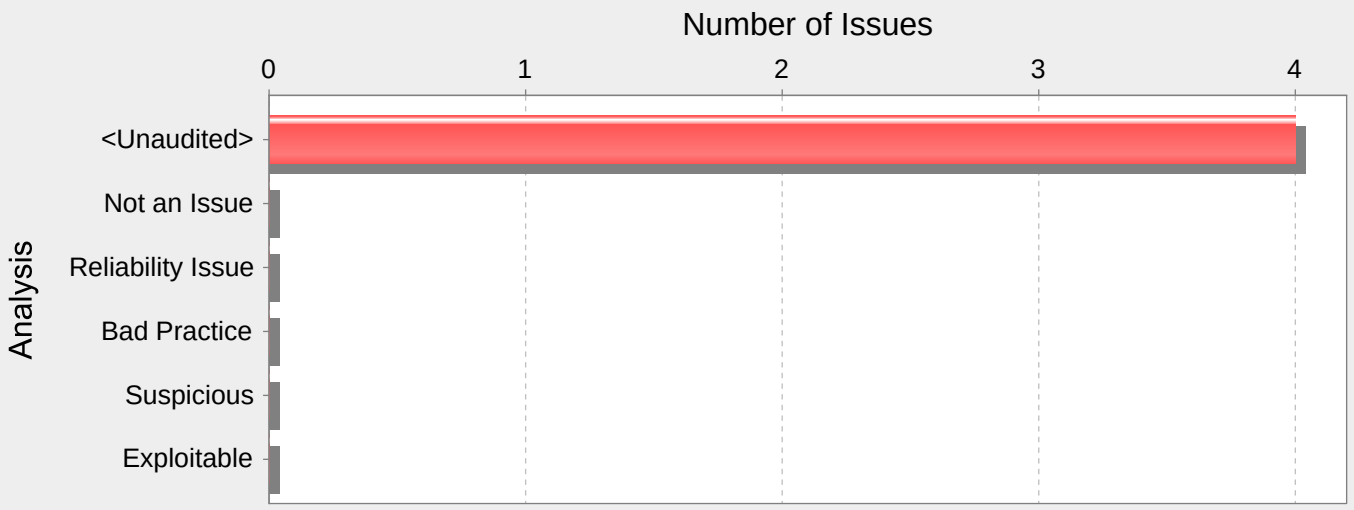
Kingdom:	Input Validation and Representation
Abstract:	Attackers are able to control the resource identifier argument to create() at AdminPanelController.java line 1335, which could enable them to access or modify otherwise protected system resources.
Source:	AdminPanelController.java:1287 javax.servlet.ServletRequest.getParameter()

```
1285    response.getWriter().append("Served at: ").append(request.getContextPath());
1286    JSONObject jsonObject;
1287    String encrString = request.getParameter("string");
1288    String handshakResponse = hsFunc(encrString);
1289    String dcrptResponse = decrptFunc(handshakResponse);
```

Sink: AdminPanelController.java:1335 java.net.URI.create()

```
1333        + ResourceBundle.getBundle("application").getString("SERVICE");
1334    HttpClient client = HttpClient.newHttpClient();
1335    HttpRequest request = HttpRequest.newBuilder().uri(URI.create(url)).build();
1336
1337    HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
```

Category: Cookie Security: HTTPOnly not Set (4 Issues)



Abstract:

The program creates a cookie in AdminPanelController.java on line 216, but fails to set the HttpOnly flag to true.

Explanation:

All major browsers support the HttpOnly cookie property that prevents client-side scripts from accessing the cookie. Cross-site scripting attacks often access cookies in an attempt to steal session identifiers or authentication tokens. Without HttpOnly enabled, attackers have easier access to user cookies.

Example 1: The following code creates a cookie without setting the HttpOnly property.

```
javax.servlet.http.Cookie cookie = new javax.servlet.http.Cookie("emailCookie", email);
// Missing a call to: cookie.setHttpOnly(true);
```

Recommendations:

Enable the HttpOnly property when you create cookies. Do this by calling, in the case of javax.servlet.http.Cookie, the setHttpOnly(boolean) method with the argument true.

Example 2: The following code creates the same cookie as the code in Example 1, but this time sets the HttpOnly parameter to true.

```
javax.servlet.http.Cookie cookie = new javax.servlet.http.Cookie("emailCookie", email);
cookie.setHttpOnly(true);
```

Several mechanisms to bypass setting HttpOnly to true have been developed, and therefore it is not completely effective.

AdminPanelController.java, line 224 (Cookie Security: HTTPOnly not Set)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The program creates a cookie in AdminPanelController.java on line 224, but fails to set the HttpOnly flag to true.		
Sink:	AdminPanelController.java:224 cookie4 = new Cookie(...)		

```
222          URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
session.getAttribute("browserId")),
223            StandardCharsets.UTF_8));
224        Cookie cookie4 = new Cookie("lti",
225          URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
session.getAttribute("localTokenId")),
226            StandardCharsets.UTF_8));
```

AdminPanelController.java, line 221 (Cookie Security: HTTPOnly not Set)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The program creates a cookie in AdminPanelController.java on line 221, but fails to set the HttpOnly flag to true.		
Sink:	AdminPanelController.java:221 cookie3 = new Cookie(...)		

```
219          URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
session.getAttribute("sessionId")),
```

```
220                         StandardCharsets.UTF_8));
221                     Cookie cookie3 = new Cookie("bi",
222                       URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
session.getAttribute("browserId")),
223                         StandardCharsets.UTF_8));
```

AdminPanelController.java, line 218 (Cookie Security: HTTPOnly not Set)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The program creates a cookie in AdminPanelController.java on line 218, but fails to set the HttpOnly flag to true.		

Sink: AdminPanelController.java:218 cookie2 = new Cookie(...)

```
216                     Cookie cookie1 = new Cookie("un", URLEncoder.encode(
217                       aesCrypto.encrypt(seedKey, (String) session.getAttribute("userName")),
StandardCharsets.UTF_8));
218                     Cookie cookie2 = new Cookie("si",
219                       URLEncoder.encode(aesCrypto.encrypt(seedKey, (String)
session.getAttribute("sessionId")),
220                         StandardCharsets.UTF_8));
```

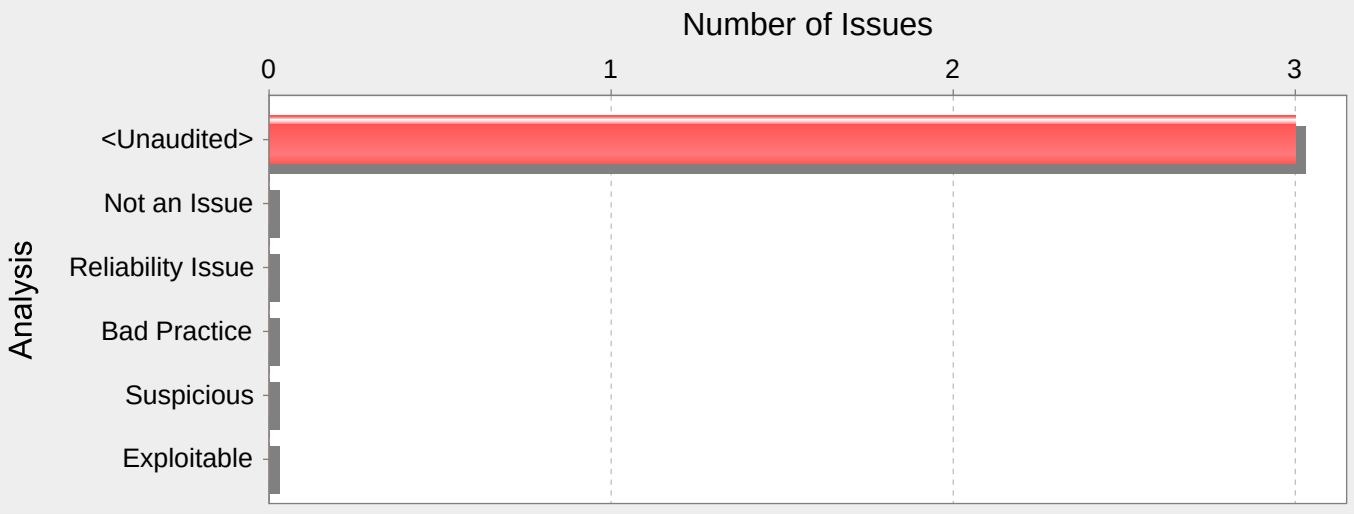
AdminPanelController.java, line 216 (Cookie Security: HTTPOnly not Set)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The program creates a cookie in AdminPanelController.java on line 216, but fails to set the HttpOnly flag to true.		

Sink: AdminPanelController.java:216 cookie1 = new Cookie(...)

```
214
215                   try {
216                     Cookie cookie1 = new Cookie("un", URLEncoder.encode(
217                       aesCrypto.encrypt(seedKey, (String) session.getAttribute("userName")),
StandardCharsets.UTF_8));
218                     Cookie cookie2 = new Cookie("si",
```

Category: Null Dereference (3 Issues)



Abstract:

The method isTokenValid() in Home.java can crash the program by dereferencing a null-pointer on line 73.

Explanation:

Null-pointer exceptions usually occur when one or more of the programmer's assumptions is violated. A dereference-after-store error occurs when a program explicitly sets an object to null and dereferences it later. This error is often the result of a programmer initializing a variable to null when it is declared.

Most null-pointer issues result in general software reliability problems, but if attackers can intentionally trigger a null-pointer dereference, they can use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example: In the following code, the programmer explicitly sets the variable foo to null. Later, the programmer dereferences foo before checking the object for a null value.

```
Foo foo = null;
...
foo.setBar(val);
...
}
```

Recommendations:

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

Home.java, line 123 (Null Dereference)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The method isTokenValidWeb() in Home.java can crash the program by dereferencing a null-pointer on line 123.		

Sink: Home.java:123 Dereferenced : responseMap()

```
121                 LOGGER.debug("parichay response fetching error. cannot convert to json format");
122               }
123               if (responseMap.get("status").equals("success")) {
124                 if (responseMap.get("tokenValid").equals("true"))
```

Home.java, line 73 (Null Dereference)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The method isTokenValid() in Home.java can crash the program by dereferencing a null-pointer on line 73.		

Sink: Home.java:73 Dereferenced : responseMap()

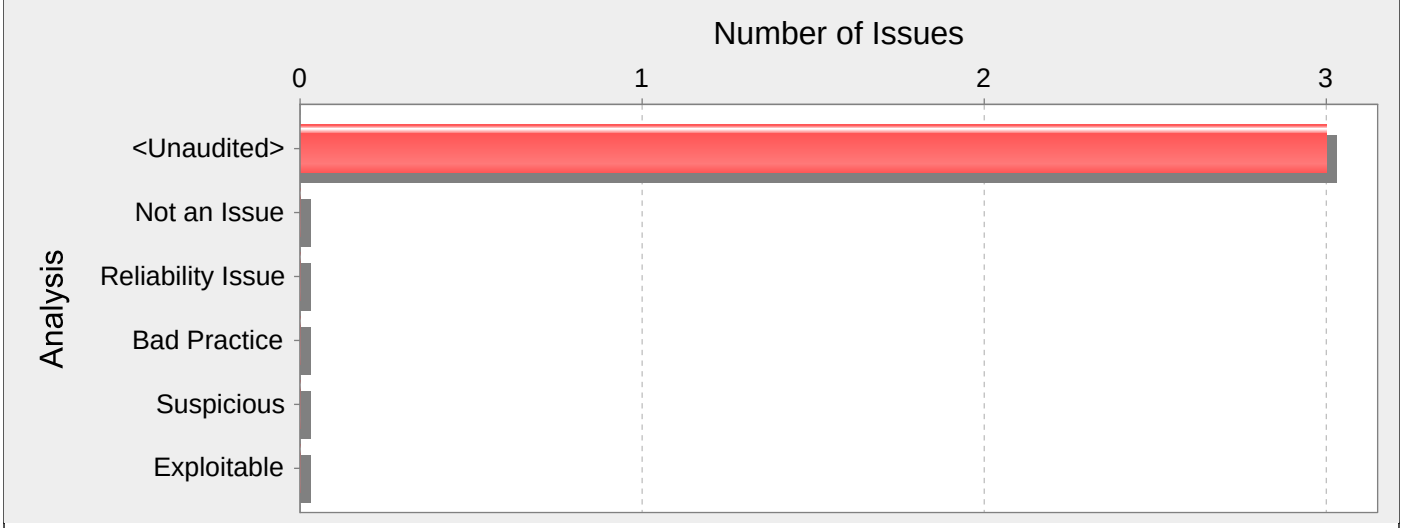
```
71                 LOGGER.debug("parichay response fetching error");
72               }
```

```
73                    if (responseMap.containsKey("status") && responseMap.get("status").equals("success"))
                    {
74                      if (responseMap.containsKey("tokenValid") &&
                    responseMap.get("tokenValid").equals("true"))
```

TokenAuth.java, line 113 (Null Dereference)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The method MailPush() in TokenAuth.java can crash the program by dereferencing a null-pointer on line 113.		
Sink:	TokenAuth.java:113 Dereferenced : token()		

```
111                   }
112
113                   String tokenstring = new String(token, 0, token.length);
114
115                   if (!localtokenid.equals(tokenstring)) {
```

Category: Poor Error Handling: Empty Catch Block (3 Issues)



Abstract:

The method getInt() in UtkalUtil.java ignores an exception on line 61, which could cause the program to overlook unexpected states and conditions.

Explanation:

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.

Two dubious assumptions that are easy to spot in code are "this method call can never fail" and "it doesn't matter if this call fails". When a programmer ignores an exception, they implicitly state that they are operating under one of these assumptions.

Example 1: The following code excerpt ignores a rarely-thrown exception from doExchange().

```
try {
doExchange();
}
catch (RareException e) {
// this can never happen
}
```

If a RareException were to ever be thrown, the program would continue to execute as though nothing unusual had occurred. The program records no evidence indicating the special situation, potentially frustrating any later attempt to explain the program's behavior.

Recommendations:

At a minimum, log the fact that the exception was thrown so that it will be possible to come back later and make sense of the resulting program behavior. Better yet, abort the current operation. If the exception is being ignored because the caller cannot properly handle it but the context makes it inconvenient or impossible for the caller to declare that it throws the exception itself, consider throwing a RuntimeException or an Error, both of which are unchecked exceptions. As of JDK 1.4, RuntimeException has a constructor that makes it easy to wrap another exception.

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

```
try {
doExchange();
}
catch (RareException e) {
throw new RuntimeException("This can never happen", e);
}
```

Tips:

1. There are rare types of exceptions that can be discarded in some contexts. For instance, Thread.sleep() throws InterruptedException, and in many situations the program should behave the same way whether or not it was awoken prematurely.

```
try {
Thread.sleep(1000);
}
```

```
catch (InterruptedException e){
// The thread has been woken up prematurely, but its
// behavior should be the same either way.
}
```

UtkalUtil.java, line 83 (Poor Error Handling: Empty Catch Block)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method encodeString() in UtkalUtil.java ignores an exception on line 83, which could cause the program to overlook unexpected states and conditions.		
Sink:	UtkalUtil.java:83 CatchBlock()		

```
81    try {
82     str = URLEncoder.encode(str, "UTF-8");
83    } catch (UnsupportedEncodingException ignored) {
84
85    }
```

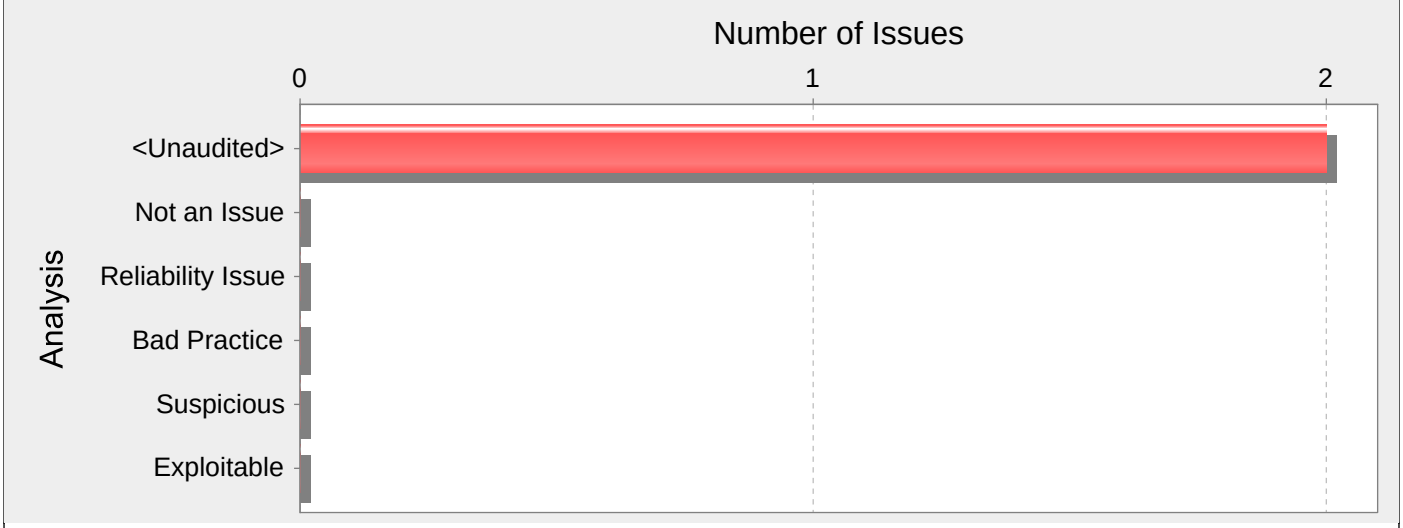
UtkalUtil.java, line 61 (Poor Error Handling: Empty Catch Block)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getInt() in UtkalUtil.java ignores an exception on line 61, which could cause the program to overlook unexpected states and conditions.		
Sink:	UtkalUtil.java:61 CatchBlock()		

```
59    try {
60     no = Integer.parseInt(str);
61    } catch (Exception e) {
62    }
63    return no;
```

UtkalUtil.java, line 74 (Poor Error Handling: Empty Catch Block)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getDouble() in UtkalUtil.java ignores an exception on line 74, which could cause the program to overlook unexpected states and conditions.		
Sink:	UtkalUtil.java:74 CatchBlock()		

```
72    try {
73     no = Double.parseDouble(str);
74    } catch (Exception e) {
75    }
76    return no;
```

Category: Code Correctness: Byte Array to String Conversion (2 Issues)



Abstract:

The call to String() on line 113 of TokenAuth.java converts a byte array into a String, which may lead to data loss.

Explanation:

When data from a byte array is converted into a String, it is unspecified what will happen to any data that is outside of the applicable character set. This can lead to data being lost, or a decrease in the level of security when binary data is needed to ensure proper security measures are followed.

Example 1: The following code converts data into a String in order to create a hash.

```
...
FileInputStream fis = new FileInputStream(myFile);
byte[] byteArr = byte[BUFSIZE];
...
int count = fis.read(byteArr);
...
String fileString = new String(byteArr);
String fileSHA256Hex = DigestUtils.sha256Hex(fileString);
// use fileSHA256Hex to validate file
...
```

Assuming the size of the file is less than BUFSIZE, this works fine as long as the information in myFile is encoded the same as the default character set, however if it's using a different encoding, or is a binary file, it will lose information. This in turn will cause the resulting SHA hash to be less reliable, and could mean it's far easier to cause collisions, especially if any data outside of the default character set is represented by the same value, such as a question mark.

Recommendations:

Generally speaking, a byte array potentially containing noncharacter data should never be converted into a String object as it may break functionality, but in some cases this can cause much larger security concerns. In a lot of cases there is no need to actually convert a byte array into a String, but if there is a specific reason to be able to create a String object from binary data, it must first be encoded in a way such that it will fit into the default character set.

Example 2: The following uses a different variant of the API in Example 1 to prevent any validation problems.

```
...
FileInputStream fis = new FileInputStream(myFile);
byte[] byteArr = byte[BUFSIZE];
...
int count = fis.read(byteArr);
...
byte[] fileSHA256 = DigestUtils.sha256(byteArr);
// use fileSHA256 to validate file, comparing hash byte-by-byte.
...
```

In this case, it is straightforward to rectify, since this API has overloaded variants including one that accepts a byte array, and this could be simplified even further by using another overloaded variant of DigestUtils.sha256() that accepts a FileInputStream object as its argument. Other scenarios may need careful consideration as to whether it's possible that the byte array could contain data outside of the character set, and further refactoring may be required.

TokenAuth.java, line 113 (Code Correctness: Byte Array to String Conversion)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The call to String() on line 113 of TokenAuth.java converts a byte array into a String, which may lead to data loss.		
Sink:	TokenAuth.java:113 String()		

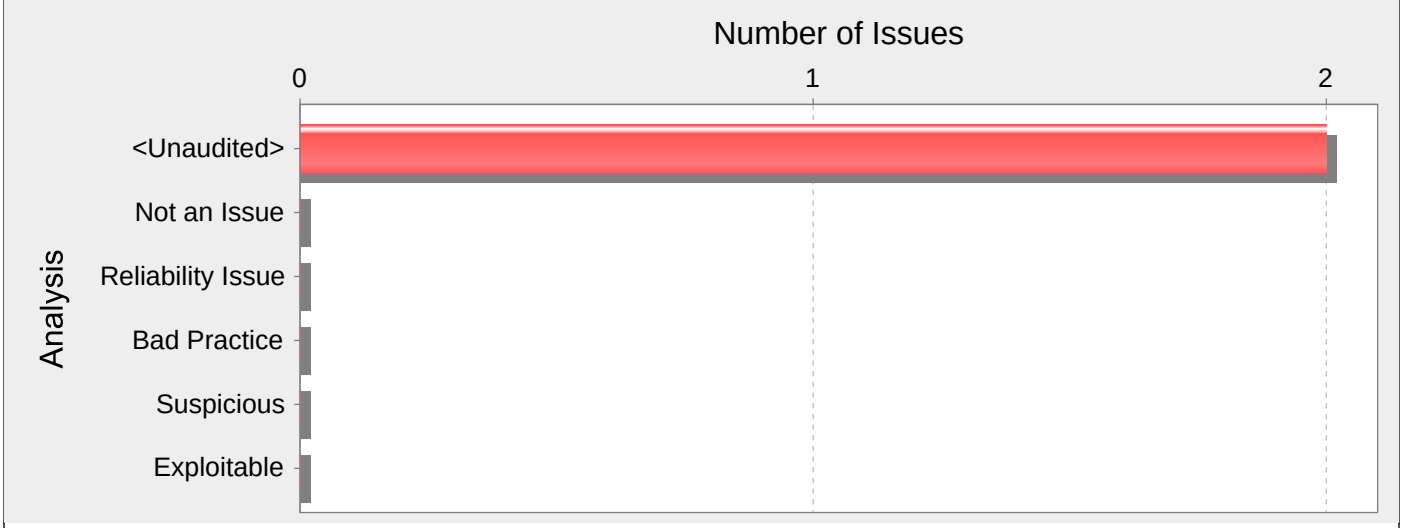
```
111   }
112
113   String tokenstring = new String(token, 0, token.length);
114
115   if (!localtokenid.equals(tokenstring)) {
```

AESEncryption.java, line 81 (Code Correctness: Byte Array to String Conversion)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The call to String() on line 81 of AESEncryption.java converts a byte array into a String, which may lead to data loss.		
Sink:	AESEncryption.java:81 String()		

```
79   byte[] stringKey = Base64.encodeBase64(sk.getEncoded());
80   LOGGER.debug("actual secret_key in string form:" + new String(stringKey,
StandardCharsets.UTF_8));
81   return new String(stringKey);
82  }
```

Category: Password Management (2 Issues)



Abstract:

The method getS3Client() in ObjStoreConfig.java uses a plain text password on line 34. Storing a password in plain text can result in a system compromise.

Explanation:

Password management issues occur when a password is stored in plain text in an application's properties or configuration file.

Example 1: The following code reads a password from a properties file and uses the password to connect to a database.

```
...
Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
String password = prop.getProperty("password");
DriverManager.getConnection(url, usr, password);
...
```

This code will run successfully, but anyone who has access to config.properties can read the value of password. Any devious employee with access to this information can use it to break into the system.

In the mobile environment, password management is especially important given that there is such a high chance of device loss.

Example 2: The following code reads username and password from an Android WebView store and uses them to setup authentication for viewing protected pages.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
String[] credentials = view.getHttpAuthUsernamePassword(host, realm);
String username = credentials[0];
String password = credentials[1];
handler.proceed(username, password);
}
});
...
```

By default, WebView credentials are stored in plain text and are not hashed. So if a user has a rooted device (or uses an emulator), she is able to read stored passwords for given sites.

Recommendations:

A password should never be stored in plain text. An administrator should be required to enter the password 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. At the very least, passwords should be hashed before being stored.

Some third-party products claim the ability to securely manage passwords. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. Today, the best option for a secure generic solution is to create a proprietary mechanism yourself.

For Android, as well as any other platform that uses SQLite database, SQLCipher is a good alternative. SQLCipher is an extension to the SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database.

Example 3: The following code demonstrates how to integrate SQLCipher into an Android application after downloading the necessary binaries, and store credentials into the database file.

```
import net.sqlcipher.database.SQLiteDatabase;
...
SQLiteDatabase.loadLibs(this);
File dbFile = getDatabasePath("credentials.db");
dbFile.mkdirs();
dbFile.delete();
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(dbFile, "credentials", null);
db.execSQL("create table credentials(u, p)");
db.execSQL("insert into credentials(u, p) values(?, ?)", new Object[]{username, password});
...
```

Note that references to android.database.sqlite.SQLiteDatabase are substituted with those of net.sqlcipher.database.SQLiteDatabase.

To enable encryption on the WebView store, you must recompile WebKit with the sqlcipher.so library.

Tips:

1. The Fortify Secure Coding Rulepacks identify password management issues by looking for functions that are known to take passwords as arguments. If a password is provided from outside the program and is used without passing through an identified de-obfuscation routine, then Fortify Static Code Analyzer flags a password management issue.

To audit a password management issue, trace through the program starting from where the password enters the system and ending where it is used. Look for code that performs de-obfuscation. If no such code is present, then this issue has not been mitigated. If the password passes through a de-obfuscation function, verify that the algorithm used to protect the password is sufficiently robust.

After you are convinced that the password is adequately protected, write a custom passthrough rule for the de-obfuscation routine that indicates that the password is protected with obfuscation. If you include this rule in future analyses of the application, passwords that pass through the identified de-obfuscation routine will no longer trigger password management vulnerabilities.

2. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Struts 2). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the 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.

MailAuthSMTP.java, line 58 (Password Management)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method getPasswordAuthentication() in MailAuthSMTP.java uses a plain text password on line 58. Storing a password in plain text can result in a system compromise.		

Source: MailAuthSMTP.java:24 java.lang.System.getenv()

```
22    private static String SMTP_AUTH_USER = System.getenv("email_username"); // user email address
23
24    private static String SMTP_AUTH_PWD = System.getenv("email_password");// Password
25    private static final Logger LOGGER = LogManager.getLogger(MailAuthSMTP.class);
```

Sink: MailAuthSMTP.java:58 javax.mail.PasswordAuthentication.PasswordAuthentication()

```
56      String username = SMTP_AUTH_USER;
57      String password =******
58      return new PasswordAuthentication(username, password);
59    }
60   }
```

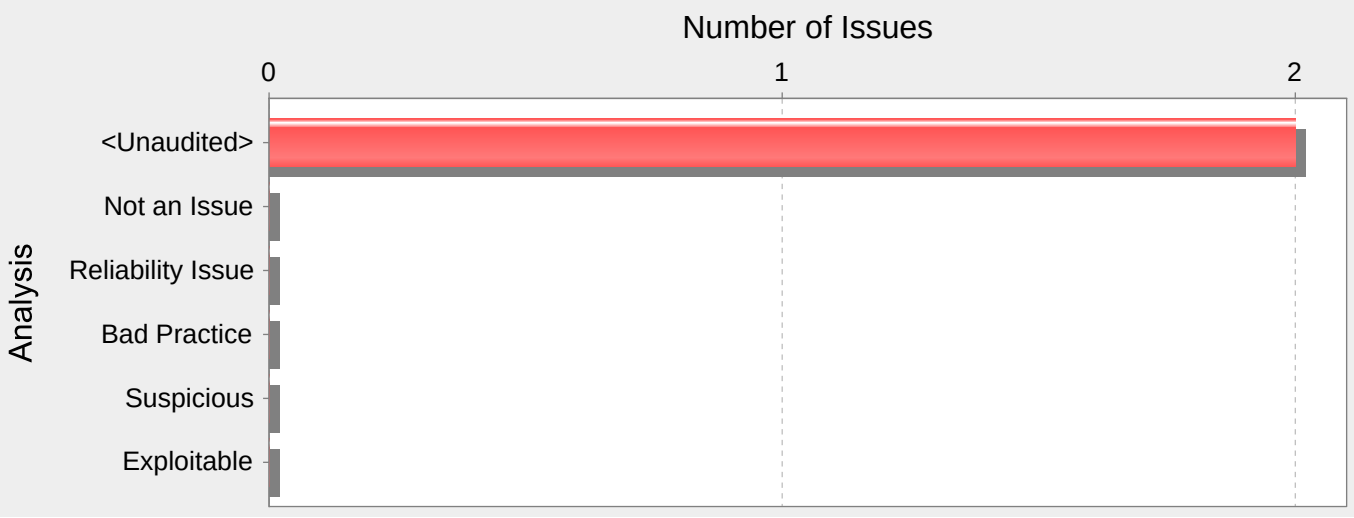
ObjStoreConfig.java, line 34 (Password Management)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method getS3Client() in ObjStoreConfig.java uses a plain text password on line 34. Storing a password in plain text can result in a system compromise.		
Source:	ObjStoreConfig.java:27 java.lang.System.getenv()		

```
25  private static final String objDirName = System.getenv("objDirName");// Specific directory in which files will be stored in ObjectStore
26  private static final String accessKey = System.getenv("accessKey");// ObjectStore accessKey
27  private static final String secretKey = System.getenv("secretKey");// ObjectStore secretKey
28  private static final String bucketName = System.getenv("bucketName");// BucketName
29  private static final String endPoint = System.getenv("staasEndPoint");
```

Sink:	ObjStoreConfig.java:34 com.amazonaws.auth.BasicAWSCredentials.BasicAWSCredentials()

```
32
33  private static AmazonS3 getS3Client() {
34    AWSCredentials credentials = new BasicAWSCredentials(accessKey, secretKey);
35    ClientConfiguration clientConfiguration = new ClientConfiguration();
36    clientConfiguration.setSignerOverride("AWSS3V4SignerType");
```

Category: Password Management: Password in Configuration File (2 Issues)



Abstract:

Storing a plain text password in a configuration file may result in a system compromise.

Explanation:

Storing a plain text 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 plain text.

Recommendations:

A password should never be stored in plain text. An administrator should be required to enter the password 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.

Some third-party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution the only viable option is a proprietary one.

Tips:

1. 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 plain text.

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.

application.properties, line 10 (Password Management: Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Storing a plain text password in a configuration file may result in a system compromise.		

Sink: application.properties:10 spring.datasource.password()

```
8  spring.datasource.url=${ndcbbsr_postgre_db_url}
9  spring.datasource.username=${ndcbbsr_postgre_db_username}
10 spring.datasource.password=******
11
12 server.servlet.session.cookie.http-only=true
```

application.properties, line 10 (Password Management: Password in Configuration File)

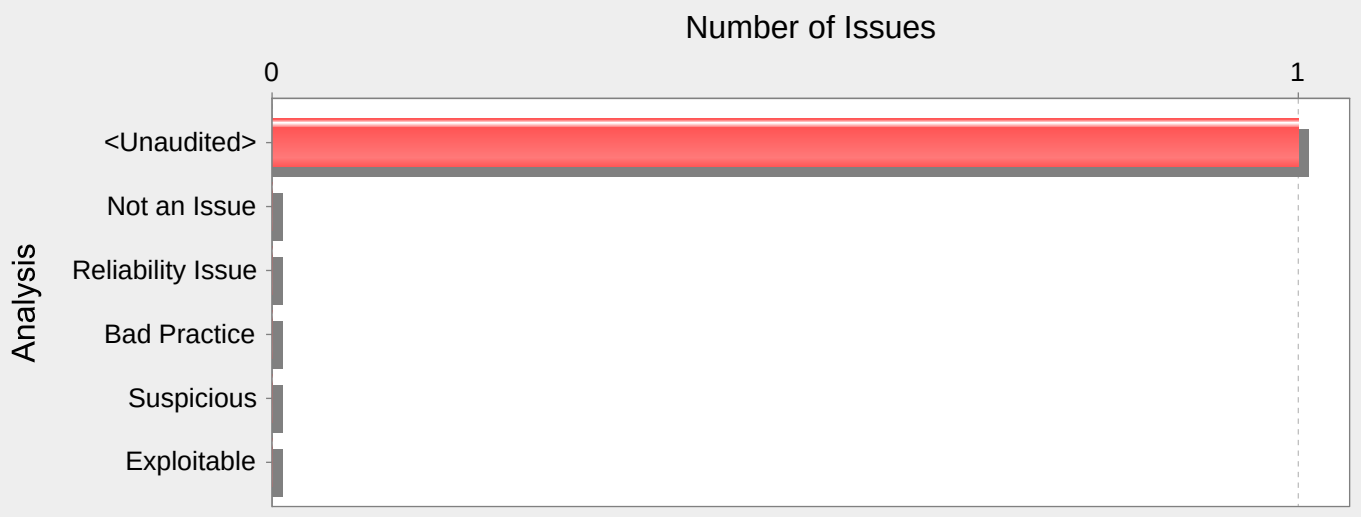
Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Storing a plain text password in a configuration file may result in a system compromise.		

Sink: application.properties:10 spring.datasource.password()

```
8  spring.datasource.url=${ndcbbsr_postgre_db_url}
9  spring.datasource.username=${ndcbbsr_postgre_db_username}
```

```
10                      spring.datasource.password=******
11
12                      server.servlet.session.cookie.http-only=true
```

Category: Cookie Security: Overly Broad Session Cookie Domain (1 Issues)



Abstract:

A session cookie with an overly broad domain can be accessed by applications sharing the same base domain.

Explanation:

Developers often set session cookies to be a base domain such as ".example.com". However, doing so exposes the session cookie to all web applications on the base domain name and any sub-domains. Leaking session cookies can lead to account compromises.

Example 1: Imagine you have a secure application deployed at http://secure.example.com/ and the application sets a session cookie with domain ".example.com" when users log in.

The application's configuration file would have the following entry:

server.servlet.session.cookie.domain=.example.com

Suppose you have another less secure application at http://insecure.example.com/ and it contains a cross-site scripting vulnerability. Any user authenticated to http://secure.example.com that browses to http://insecure.example.com risks exposing their session cookie from http://secure.example.com.

Recommendations:

Make sure to set cookie domains to be as restrictive as possible.

Example 2: The following configuration option in application.properties shows how to set the session cookie domain to "secure.example.com" for the Example 1 example.

server.servlet.session.cookie.domain=secure.example.com

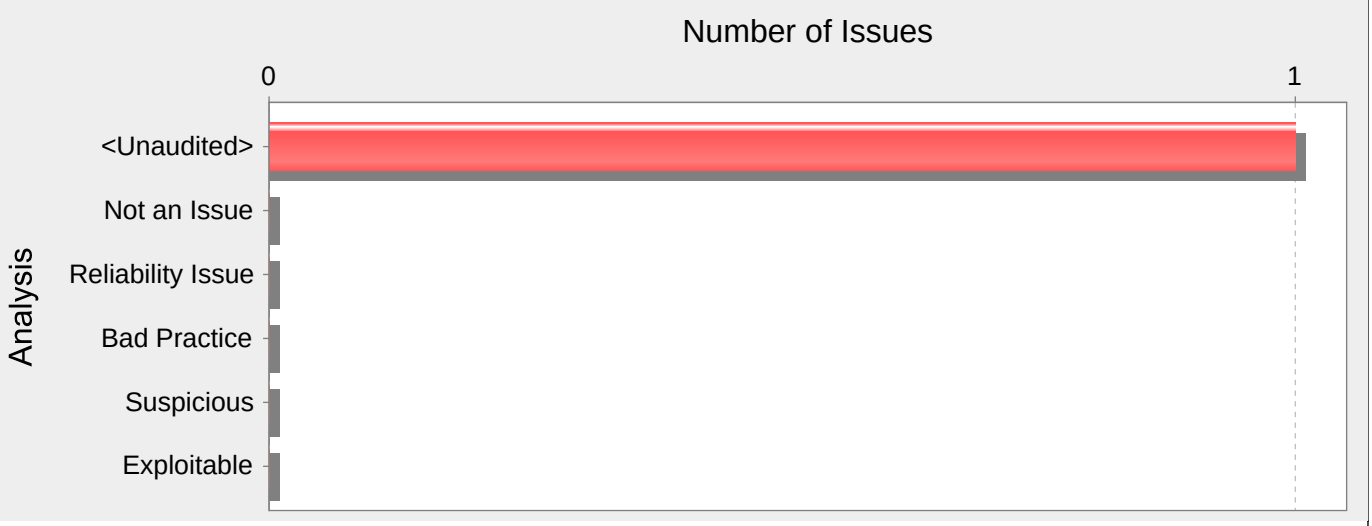
application.properties, line 32 (Cookie Security: Overly Broad Session Cookie Domain)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	A session cookie with an overly broad domain can be accessed by applications sharing the same base domain.		

Sink: application.properties:32 server.servlet.session.cookie.domain()

```
30  #spring.datasource.password=******
31
32  server.servlet.session.cookie.domain=.ndcbbsr.nic.in
33  server.servlet.session.cookie.http-only=true
34  server.servlet.session.cookie.path=/
```

Category: Cookie Security: Overly Broad Session Cookie Path (1 Issues)



Abstract:

A session cookie with an overly broad path can be compromised through applications sharing the same domain.

Explanation:

Developers often set session cookies to be the root context path ("/"). This exposes the cookie to all web applications on the same domain name. Leaking session cookies can lead to account compromises because an attacker may steal the session cookie using a vulnerability in any of the applications on the domain.

Example 1: Imagine you have a forum application deployed at http://communitypages.example.com/MyForum and the application sets a session cookie with path "/" when users log in to the forum. For example:

server.servlet.session.cookie.path=/

Suppose an attacker creates another application at http://communitypages.example.com/EvilSite and posts a link to this site on the forum. When a user of the forum clicks this link, his browser will send the session cookie set by /MyForum to the application running at /EvilSite. By using the session cookie provided from the user on /MyForum, the attacker can compromise the account of any forum user that browses to /EvilSite.

Recommendations:

Set session cookie paths to be as restrictive as possible.

Example 2: The following code shows how to set the session cookie path to "/MyForum" for Example 1.

server.servlet.session.cookie.path=/MyForum

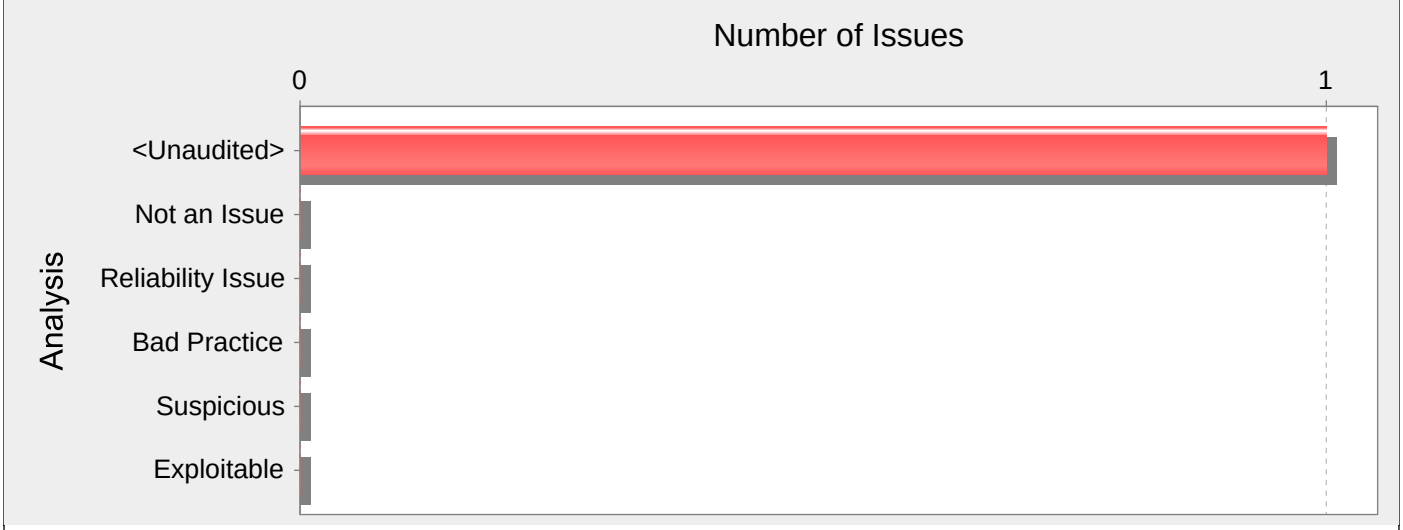
application.properties, line 34 (Cookie Security: Overly Broad Session Cookie Path)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	A session cookie with an overly broad path can be compromised through applications sharing the same domain.		

Sink: application.properties:34 server.servlet.session.cookie.path()

```
32  server.servlet.session.cookie.domain=.ndcbbsr.nic.in
33  server.servlet.session.cookie.http-only=true
34  server.servlet.session.cookie.path=/
35
36  server.servlet.session.timeout=30m
```

Category: Dead Code: Unused Method (1 Issues)



Abstract:

The method bytesToHex() in AesCrypto.java is not reachable from any method outside the class. It is dead code. Dead code is defined as code that is never directly or indirectly executed by a public method.

Explanation:

This method is never called or is only called from other dead code.

Example 1: In the following class, the method doWork() can never be called.

```
public class Dead {
private void doWork() {
System.out.println("doing work");
}
public static void main(String[] args) {
System.out.println("running Dead");
}
}
```

Example 2: In the following class, two private methods call each other, but since neither one is ever invoked from anywhere else, they are both dead code.

```
public class DoubleDead {
private void doTweedledee() {
doTweedledumb();
}
private void doTweedledumb() {
doTweedledee();
}
public static void main(String[] args) {
System.out.println("running DoubleDead");
}
}
```

(In this case it is a good thing that the methods are dead: invoking either one would cause an infinite loop.)

Recommendations:

A dead method may indicate a bug in dispatch code.

Example 3: If method is flagged as dead named getWitch() in a class that also contains the following dispatch method, it may be because of a copy-and-paste error. The 'w' case should return getWitch() not getMummy().

```
public ScaryThing getScaryThing(char st) {
switch(st) {
case 'm':
return getMummy();
```

```
case 'w':
return getMummy();
default:
return getBlob();
}
}
```

In general, you should repair or remove dead code. To repair dead code, execute the dead code directly or indirectly through a public method. Dead code causes additional complexity and maintenance burden without contributing to the functionality of the program.

Tips:

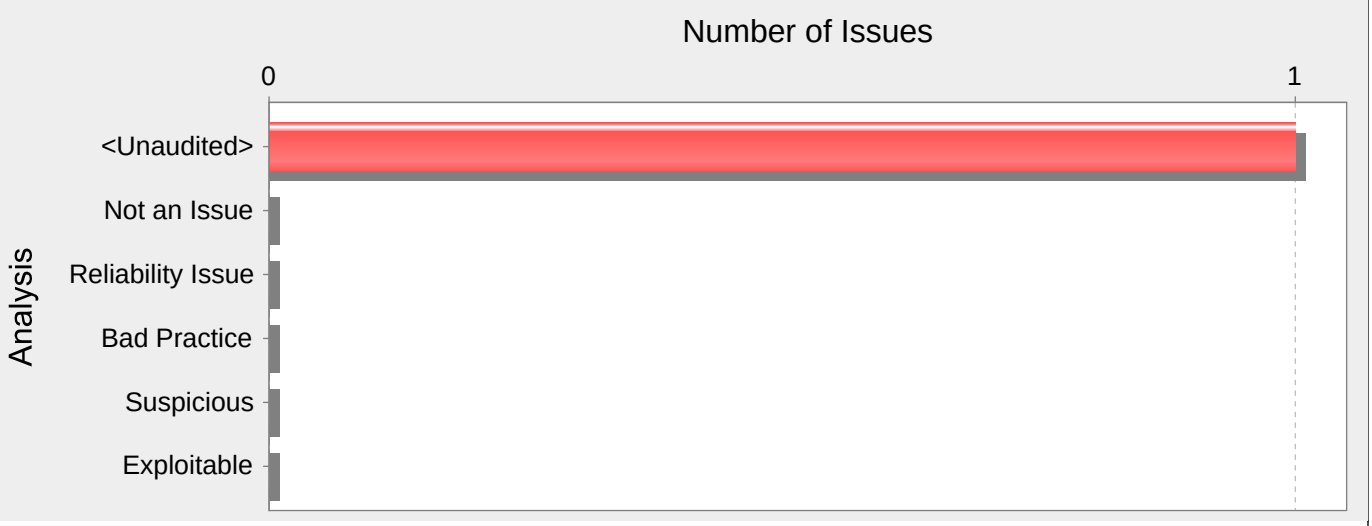
1. This issue may be a false positive if the program uses reflection to access private methods. (This is a non-standard practice. Private methods that are only invoked via reflection should be well documented.)

AesCrypto.java, line 29 (Dead Code: Unused Method)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method bytesToHex() in AesCrypto.java is not reachable from any method outside the class. It is dead code. Dead code is defined as code that is never directly or indirectly executed by a public method.		

Sink: AesCrypto.java:29 Function: bytesToHex()

```
27  private final char[] hexArray = "0123456789ABCDEF".toCharArray();
28
29  private String bytesToHex(byte[] bytes) {
30   char[] hexChars = new char[bytes.length * 2];
31   for (int j = 0; j < bytes.length; j++) {
```

Category: Header Manipulation: SMTP (1 Issues)



Abstract:

The method SendMail() in MailAuthSMTP.java includes unvalidated data in an SMTP header on line 42. This enables attackers to add arbitrary headers such as CC or BCC that they can use to leak the mail contents to themselves or use the mail server as a spam bot.

Explanation:

SMTP Header Manipulation vulnerabilities occur when:

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

2. The data is included in an SMTP header sent to a mail server without being validated.

As with many software security vulnerabilities, SMTP 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 SMTP header.

One of the most common SMTP Header Manipulation attacks is used for distributing spam emails. If an application contains a vulnerable "Contact us" form that allows setting the subject and the body of the email, an attacker will be able to set any arbitrary content and inject a CC header with a list of email addresses to spam anonymously since the email will be sent from the victim server.

Example: The following code segment reads the subject and body of a "Contact us" form:

```
String subject = request.getParameter("subject");
String body = request.getParameter("body");
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress("webform@acme.com"));
message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("support@acme.com"));
message.setSubject("[Contact us query] " + subject);
message.setText(body);
Transport.send(message);
```

Assuming a string consisting of standard alphanumeric characters, such as "Page not working" is submitted in the request, the SMTP headers might take the following form:

```
...
subject: [Contact us query] Page not working
...
```

However, because the value of the header is constructed from unvalidated user input the response will only maintain this form if the value submitted for subject does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Congratulations!! You won the lottery!!!\r\ncc:victim1@mail.com,victim2@mail.com ...", then the SMTP headers would be of the following form:

```
...
subject: [Contact us query] Congratulations!! You won the lottery
cc: victim1@mail.com,victim2@mail.com
...
```

This will effectively allow an attacker to craft spam messages or to send anonymous emails amongst other attacks.

Recommendations:

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

Since SMTP Header Manipulation vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it is used in the header context and make sure there are no illegal CRLF characters that can break the header structure.

MailAuthSMTP.java, line 42 (Header Manipulation: SMTP)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method SendMail() in MailAuthSMTP.java includes unvalidated data in an SMTP header on line 42. This enables attackers to add arbitrary headers such as CC or BCC that they can use to leak the mail contents to themselves or use the mail server as a spam bot.		

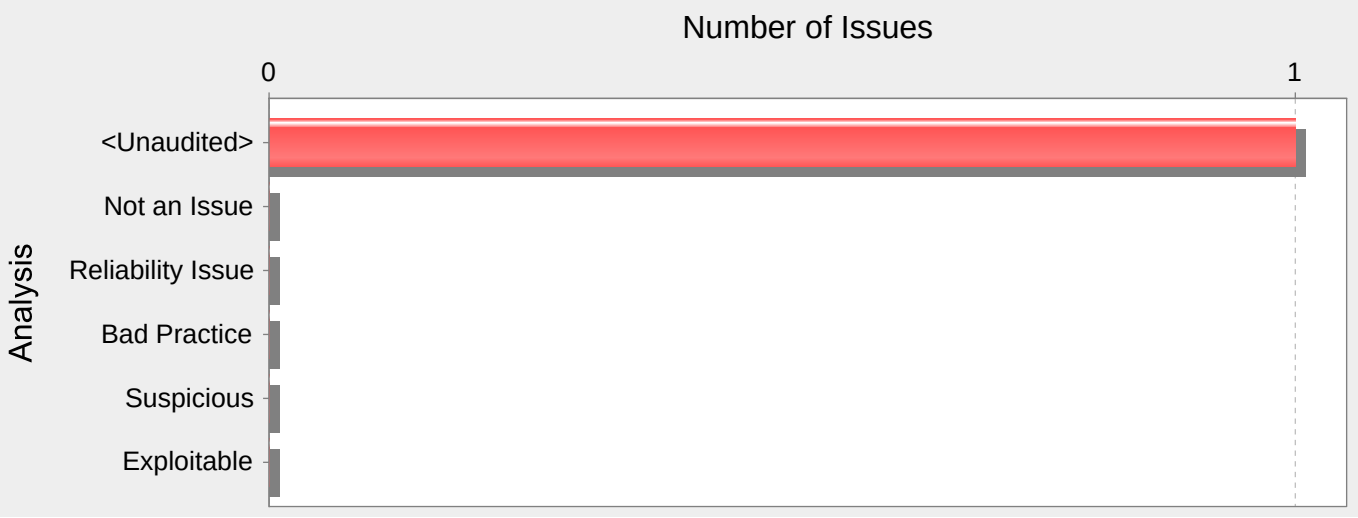
Source: TokenAuth.java:78 MailPush(3)

```
76
77    @GetMapping("/mailpush")
78    public Map<String, String> MailPush(String url, String tomailid, String content,
      String subject, String sender)
79       throws JsonProcessingException {
```

Sink: MailAuthSMTP.java:42 javax.mail.internet.MimeMessage.setSubject()

```
40       MimeMessage message = new MimeMessage(mailSession);
41       message.setContent(msg, "text/html; charset=utf-8");
42       message.setSubject(subject);
43       message.setFrom(new InternetAddress("noreply-ndcbbsr@nic.in")); // from address
44       message.addRecipient(Message.RecipientType.TO, new InternetAddress(email));
```

Category: HTML5: Missing Content Security Policy (1 Issues)



Abstract:

Content Security Policy (CSP) is not configured.

Explanation:

Content Security Policy (CSP) is a declarative security header that enables developers to dictate which domains the site is allowed to load content from or initiate connections to when rendered in the web browser. It provides an additional layer of security from critical vulnerabilities such as cross-site scripting, clickjacking, cross-origin access and the like, on top of input validation and checking an allow list in code.

Spring Security and other frameworks do not add Content Security Policy headers by default. The web application author must declare the security policy/policies to enforce or monitor for the protected resources to benefit from this additional layer of security.

Recommendations:

Configure a Content Security Policy to mitigate possible injection vulnerabilities.

Example: The following code sets a Content Security Policy in a Spring Security protected application:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
String policy = getCSPolicy();
http.headers().contentSecurityPolicy(policy);
...
}
```

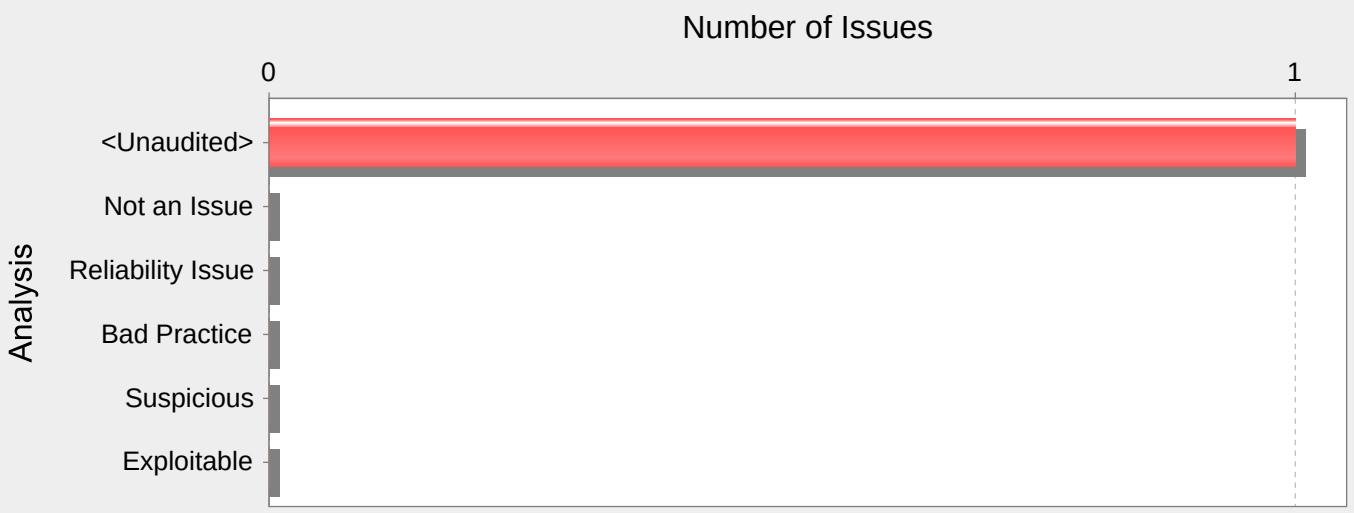
Content Security Policy is not intended to solve all content injection vulnerabilities. Instead, you can leverage CSP to help reduce the harm caused by content injection attacks. Use regular defensive coding,above, current such as input validation and output encoding.

SecSecurityConfig.java, line 23 (HTML5: Missing Content Security Policy)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Encapsulation		
Abstract:	Content Security Policy (CSP) is not configured.		
Sink:	SecSecurityConfig.java:23 Function: configure()		

```
21
22    @Override
23    protected void configure(HttpSecurity http) {
24
25      try {
```

Category: Poor Style: Value Never Read (1 Issues)



Abstract:

The method MailPush() in TokenAuth.java never uses the value it assigns to the variable sender on line 137.

Explanation:

This variable's value is not used. After the assignment, the variable is either assigned another value or goes out of scope.

Example: The following code excerpt assigns to the variable r and then overwrites the value without using it.

```
r = getName();
r = getNewBuffer(buf);
```

Recommendations:

Remove unnecessary assignments in order to make the code easier to understand and maintain.

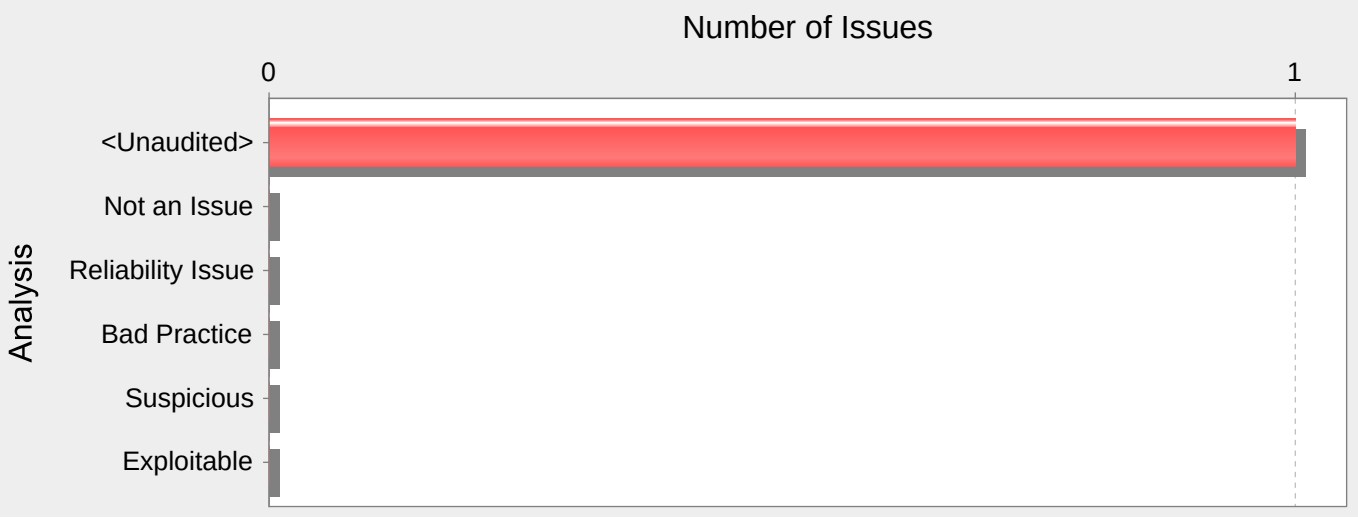
TokenAuth.java, line 137 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method MailPush() in TokenAuth.java never uses the value it assigns to the variable sender on line 137.		

Sink: TokenAuth.java:137 VariableAccess: sender()

```
135    content = UtkalUtil.safeLogMsg(250, content);
136    subject = UtkalUtil.safeLogMsg(50, subject);
137    sender = UtkalUtil.safeLogMsg(50, sender);
138
139    String ipNow = request.getRemoteAddr();
```

Category: Privacy Violation: Heap Inspection (1 Issues)



Abstract:

The method MailPush() in TokenAuth.java stores sensitive data in a String object, making it impossible to reliably purge the data from memory.

Explanation:

Sensitive data (such as passwords, social security numbers, credit card numbers etc) stored in memory can be leaked if memory is not cleared after use. Often, Strings are used store sensitive data, however, since String objects are immutable, removing the value of a String from memory can only be done by the JVM garbage collector. The garbage collector is not required to run unless the JVM is low on memory, so there is no guarantee as to when garbage collection will take place. In the event of an application crash, a memory dump of the application might reveal sensitive data.

Example 1: The following code converts a password from a character array to a String.

```
private JPasswordField pf;
...
final char[] password = pf.getPassword();
...
String passwordAsString = new String(password);
```

This category was derived from the Cigital Java Rulepack.

Recommendations:

Always be sure to clear sensitive data that is no longer needed. Instead of storing sensitive data in immutable objects such as Strings, use byte arrays or character arrays that you can clear programmatically.

Example 2: The following code clears memory after a password is used.

```
private JPasswordField pf;
...
final char[] password = pf.getPassword();
// use the password
...
// erase when finished
Arrays.fill(password, ' ');
```

Tips:

1. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the 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.

TokenAuth.java, line 113 (Privacy Violation: Heap Inspection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The method MailPush() in TokenAuth.java stores sensitive data in a String object, making it impossible to reliably purge the data from memory.		

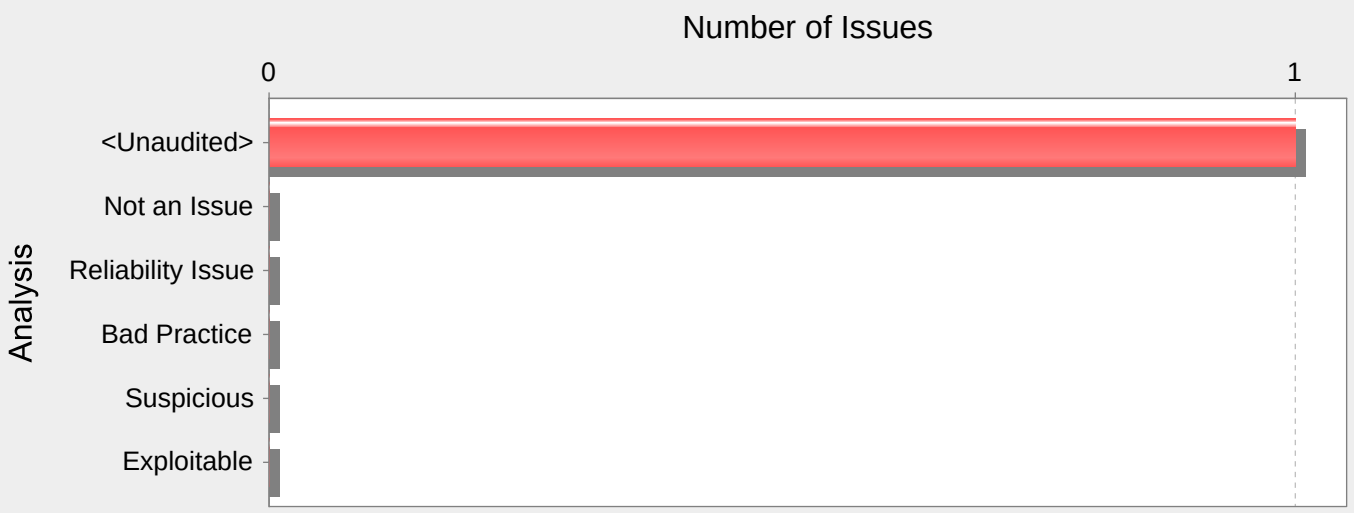
Source: TokenAuth.java:108 com.example.ndcbbsrweb.util.AesCrypto.decrypt()

```
106                 byte[] token = null;
107                 try {
108                  token = aesCrypto.decrypt(cookieValue, decryptkey);
109                 } catch (Exception e1) {
110                  LOGGER.debug("AdminPanel.homepage aesCrypto.decrypt Exception");
```

Sink: TokenAuth.java:113 java.lang.String.String()

```
111                 }
112
113                 String tokenstring = new String(token, 0, token.length);
114
115                 if (!localtokenid.equals(tokenstring)) {
```

Category: Spring Security Misconfiguration: Lack of Fallback Check (1 Issues)



Abstract:

Spring Security configuration lacks a fallback check to apply to unmatched requests.

Explanation:

Spring Security uses an expression-based access control that lets developers define a set of checks that must be applied to every request. To determine if the access control must be applied to the request, Spring Security attempts to match the request with the request matcher defined for every security check. If the request matches, the access control is applied to the request. A special request matcher exists to always match against any requests: anyRequest(). Failing to define a fallback check that uses the anyRequest() matcher, might leave endpoints unprotected.

Example 1: The following code defines a Spring Security configuration that fails to define a fallback check:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
http.authorizeRequests()
.mvcMatchers("/admin").hasRole("ADMIN");
...
}
```

In the previous Example 1 example, current or future endpoints such as /admin/panel might be left unprotected.

Recommendations:

As a security best practice, always include a catch-all matcher that denies access to any previously unmatched requests.

Example 2: The following code defines a Spring Security configuration that defaults to deny access to any unmatched requests:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
http.authorizeRequests()
.mvcMatchers("/admin").hasRole("ADMIN")
.mvcMatchers("/home").anonymous()
.anyRequest().denyAll();
...
}
```

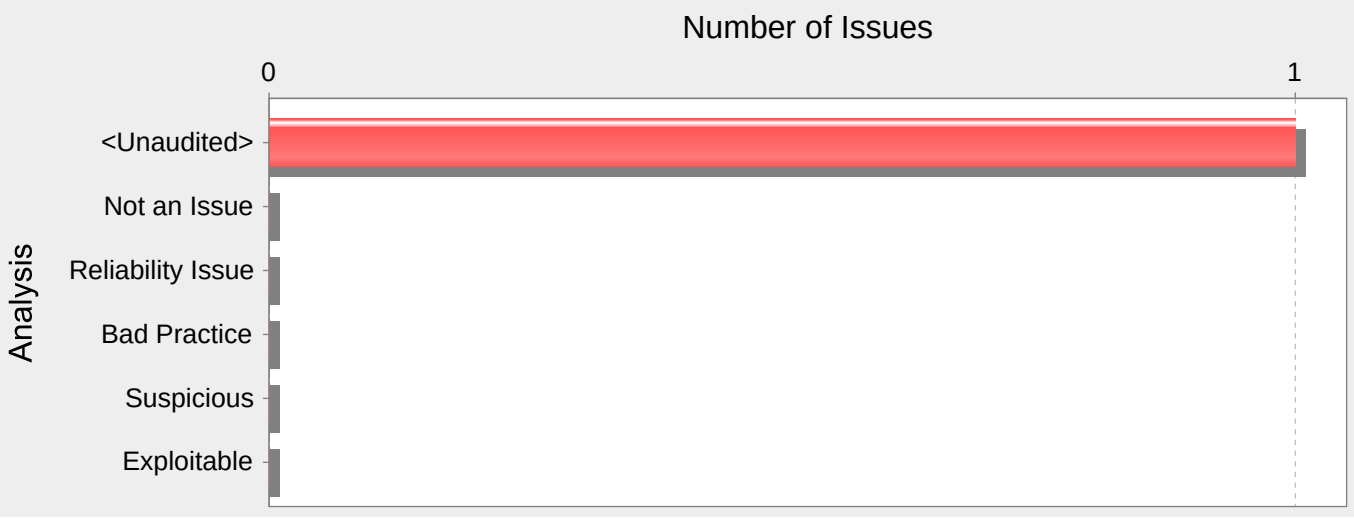
SecSecurityConfig.java, line 23 (Spring Security Misconfiguration: Lack of Fallback Check)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Spring Security configuration lacks a fallback check to apply to unmatched requests.		
Sink:	SecSecurityConfig.java:23 Function: configure()		

```
21
22    @Override
23    protected void configure(HttpSecurity http) {
```

```
24
25                      try {
```

Category: Weak Cryptographic Hash: User-Controlled PBE Salt (1 Issues)



Abstract:

The function decrypt() in AesCrypto.java includes user input within the salt value used within a Password-Based Key Derivation Function (PBKDF) on line 68. This may enable the attacker to specify an empty salt, allowing for both more easily determined hashed values and a leak of information about how the program performs its cryptographic hashing.

Explanation:

Weak Cryptographic Hash: User-Controlled PBE Salt issues occur when:

1. Data enters a program through an untrusted source

2. User-controlled data is included within the salt, or used entirely as the salt within a Password-Based Key Derivation Function (PBKDF).

As with many software security vulnerabilities, Weak Cryptographic Hash: User-Controlled PBE Salt is a means to an end, not an end in and of itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to an application, and the data is then used as all or part of the salt in a PBKDF.

The problem with having a user-defined salt is that it can enable various attacks:

1. The attacker may use this vulnerability to specify an empty salt for their own password. From this, it would be trivial to quickly derive their own password using a number of different password-based key derivation functions to leak information about the PBKDF implementation used within your application. This could make "cracking" other passwords easier by being able to limit the particular variant of hash used.

2. If the attacker is able to manipulate other users' salts, or trick other users into using an empty salt, this would enable them to compute "rainbow tables" for the application and more easily determine the derived values.

Example 1: The following code uses a user-controlled salt:

```
...
Properties prop = new Properties();
prop.load(new FileInputStream("local.properties"));
String salt = prop.getProperty("salt");
...
PBEKeySpec pbeSpec=new PBEKeySpec(password);
SecretKeyFactory keyFact=SecretKeyFactory.getInstance(CIPHER_ALG);
PBEParameterSpec defParams=new PBEParameterSpec(salt,0);
Cipher cipher=Cipher.getInstance(CIPHER_ALG);
cipher.init(cipherMode,keyFact.generateSecret(pbeSpec),defParams);
...
```

The code in Example 1 will run successfully, but anyone who can get to this functionality will be able to manipulate the salt used to derive the key or password by modifying the property salt. After the program ships, it can be nontrivial to undo an issue regarding user-controlled salts, as it is extremely difficult to know if a malicious user determined the salt of a password hash.

Recommendations:

The salt should never be user-controlled, even partially, nor hardcoded. Generally it should be obfuscated and managed in an external source. Storing a salt in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the salt.

AesCrypto.java, line 68 (Weak Cryptographic Hash: User-Controlled PBE Salt)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The function decrypt() in AesCrypto.java includes user input within the salt value used within a Password-Based Key Derivation Function (PBKDF) on line 68. This may enable the attacker to specify an empty salt, allowing for both more easily determined hashed values and a leak of information about how the program performs its cryptographic hashing.		

Source: TokenAuth.java:96 org.springframework.web.util.WebUtils.getCookie()

```
94    String localtokenid = (String) session.getAttribute("localTokenId");
95
96    Cookie emailAuthCookie = WebUtils.getCookie(request, "emailAuth");
97
98    String cookieValue = emailAuthCookie.getValue();
```

Sink: AesCrypto.java:68 javax.crypto.spec.PBEKeySpec.PBEKeySpec()

```
66     byte[] encrypted = new byte[cipherMessage.length - IV_LENGTH_BYTE];
67     System.arraycopy(cipherMessage, 0, iv, 0, iv.length);
68     PBEKeySpec pbeKeySpec = new PBEKeySpec(keyString.toCharArray(), iv, 200_000, 128);
69     SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
70     SecretKey pbeKey = factory.generateSecret(pbeKeySpec);
```