Fortify Developer Workbook

Jun 23, 2021

PruAdmin

Report Overview

Report Summary

On Jun 23, 2021, a source code review was performed over the PRUFastMobile code base. 824 files, 126,922 LOC (Executable) were scanned. A total of 112 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of issues found in this project. Specific examples and source code are provided for each issue type.

Issues by Fortify Priority Order	
Low	112

Issue Summary

Overall number of results

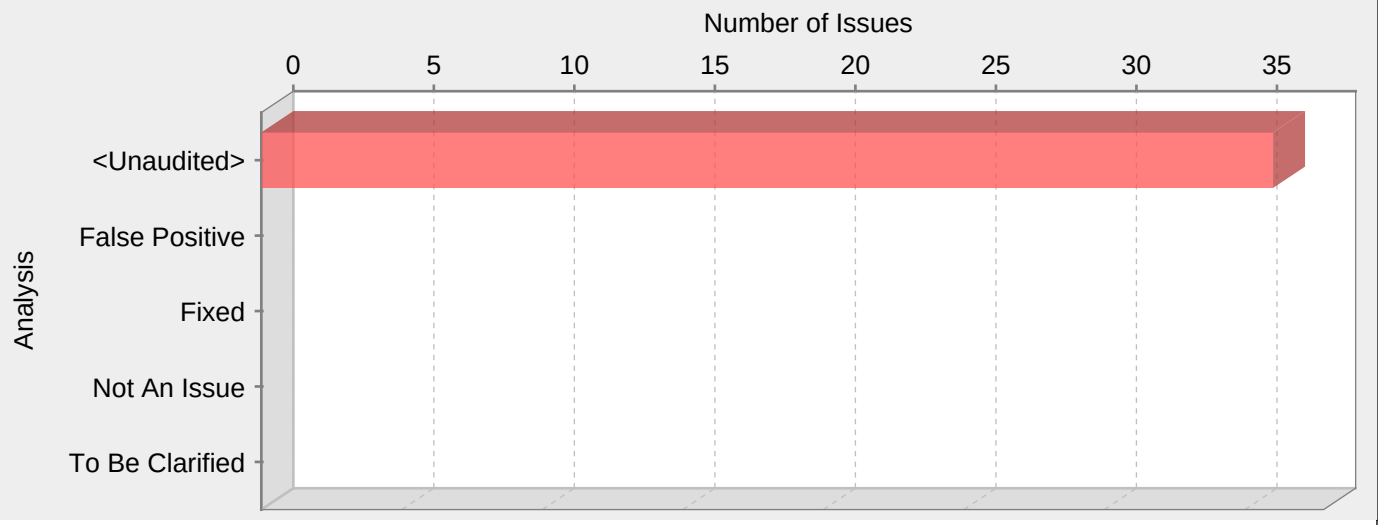
The scan found 112 issues.

Issues by Category	
System Information Leak: Internal	36
Cross-Site Request Forgery	32
Weak Cryptographic Hash	32
Password Management: Password in Comment	8
Password Management: Null Password	2
SQL Injection	2

Results Outline

Vulnerability Examples by Category

Category: System Information Leak: Internal (36 Issues)



Abstract:

The function lambda() in VideoPlayer.js might reveal system data or debugging information by calling log() on line 27. The information revealed by log() could help an adversary form a plan of attack.

Explanation:

An internal information leak occurs when system data or debugging information is sent to a local file, console, or screen via printing or logging.

Example 1: The following code prints an exception to the standard error stream:

```
var http = require('http');
...
http.request(options, function(res){
...
}).on('error', function(e){
console.log('There was a problem with the request: ' + e);
});
...
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendations:

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Tips:

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

VideoPlayer.js, line 27 (System Information Leak: Internal)

Fortify Priority: Low **Folder** Low

Kingdom:	Encapsulation
Abstract:	The function lambda() in VideoPlayer.js might reveal system data or debugging information by calling log() on line 27. The information revealed by log() could help an adversary form a plan of attack.

Source: VideoPlayer.js:27 lambda(0)

```
25        .fetch('GET', uri)
26        .then(res => Alert.alert(_('Download'), `${_('File berhasil disimpan di')} ${res.path()}`))
27        .catch(err => console.log(_('Gagal menyimpan data dari '), uri, `[${err}]`));
28      }
```

Sink: VideoPlayer.js:27 ~JS_Generic.log()

```
25        .fetch('GET', uri)
26        .then(res => Alert.alert(_('Download'), `${_('File berhasil disimpan di')} ${res.path()}`))
27        .catch(err => console.log(_('Gagal menyimpan data dari '), uri, `[${err}]`));
28      }
```

index.js, line 249 (System Information Leak: Internal)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function lambda() in index.js might reveal system data or debugging information by calling log() on line 249. The information revealed by log() could help an adversary form a plan of attack.		

Source: index.js:248 lambda(0)

```
246             });
247         })
248         .catch((onRejected) => {
249           console.log('error', onRejected);
250         });
```

Sink: index.js:249 ~JS_Generic.log()

```
247         })
248         .catch((onRejected) => {
249           console.log('error', onRejected);
250         });
```

Documents.js, line 84 (System Information Leak: Internal)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function lambda() in Documents.js might reveal system data or debugging information by calling log() on line 84. The information revealed by log() could help an adversary form a plan of attack.		

Source: Documents.js:84 lambda(0)

```
82          return RNFetchBlob.fs.readFile(source, encodingType.BASE64)
83            .then(str => this.props.onImageTaken(str, id, true))
84            .catch((err) => { Alert.alert('', _('Tidak dapat memilih video')); console.log('RNFetchBlob err: ', err); });
85        });
86      });
```

Sink: Documents.js:84 ~JS_Generic.log()

```
82          return RNFetchBlob.fs.readFile(source, encodingType.BASE64)
83            .then(str => this.props.onImageTaken(str, id, true))
84            .catch((err) => { Alert.alert('', _('Tidak dapat memilih video')); console.log('RNFetchBlob err: ', err); });
85        });
86      });
```

index.js, line 245 (System Information Leak: Internal)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		

Abstract:	The function lambda() in index.js might reveal system data or debugging information by calling log() on line 245. The information revealed by log() could help an adversary form a plan of attack.
Source:	index.js:244 lambda(0)

```
242                     });
243                 })
244                 .catch((onRejected) => {
245                   console.log('reason why doSave at initialLoad rejected is: ',
onRejected);
246                 });
```

Sink:	index.js:245 ~JS_Generic.log()

```
243                 })
244                 .catch((onRejected) => {
245                   console.log('reason why doSave at initialLoad rejected is: ',
onRejected);
246                 });
247             })
```

index.js, line 241 (System Information Leak: Internal)

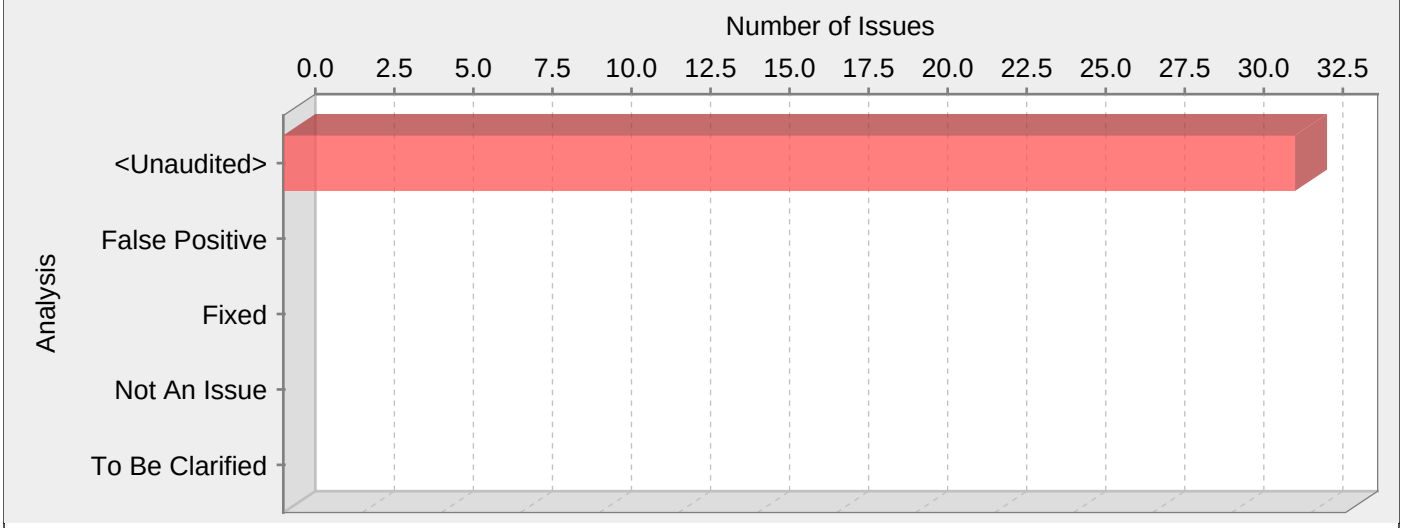
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The function lambda() in index.js might reveal system data or debugging information by calling log() on line 241. The information revealed by log() could help an adversary form a plan of attack.		
Source:	index.js:240 lambda(0)		

```
238                   }
239                 })
240                 .catch((onRejected) => {
241                   console.log('problem at initialLoad with error: ', onRejected);
242                 });
```

Sink:	index.js:241 ~JS_Generic.log()

```
239                 })
240                 .catch((onRejected) => {
241                   console.log('problem at initialLoad with error: ', onRejected);
242                 });
243                 })
```

Category: Cross-Site Request Forgery (32 Issues)



Abstract:

The HTTP request at UserProfileService.js line 22 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.

Explanation:

A cross-site request forgery (CSRF) vulnerability occurs when:

1. A web application uses session cookies.

2. The application acts on an HTTP request without verifying that the request was made with the user's consent.

A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a web application that allows administrators to create new accounts as follows:

```
var req = new XMLHttpRequest();
req.open("POST", "/new_user", true);
body = addToPost(body, new_username);
body = addToPost(body, new_passwd);
req.send(body);
```

An attacker might set up a malicious web site that contains the following code.

```
var req = new XMLHttpRequest();
req.open("POST", "http://www.example.com/new_user", true);
body = addToPost(body, "attacker");
body = addToPost(body, "haha");
req.send(body);
```

If an administrator for example.com visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application.

Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request.

CSRF is entry number five on the 2007 OWASP Top 10 list.

Recommendations:

Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, like this:

```
RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
body = addToPost(body, new_username);
body = addToPost(body, new_passwd);
```

body = addToPost(body, request_id);
rb.sendRequest(body, new NewAccountCallback(callback));

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using SSLv3.

Additional mitigation techniques include:

Framework protection: Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens.

Use a Challenge-Response control: Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens.

Check HTTP Referer/Origin headers: An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks.

Double-submit Session Cookie: Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy.

Limit Session Lifetime: When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack.

The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

Tips:

1. SCA flags all HTML forms and all XMLHttpRequest objects that might perform a POST operation. The auditor must determine if each form could be valuable to an attacker as a CSRF target and whether or not an appropriate mitigation technique is in place.

UserProfileService.js, line 44 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The HTTP request at UserProfileService.js line 44 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	UserProfileService.js:44 AssignmentStatement()		

```
42      const requestObject = {
43        adapter: `${adapterEmergencyContact}/addEmergencycontact`,
44        method: 'post',
45        parameters: { params: `['${agentNumber}','${contactName}','${relation}','${phoneNumber}','${address1}','${address2}','${address3}']` },
46      };
```

UserProfileService.js, line 66 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The HTTP request at UserProfileService.js line 66 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	UserProfileService.js:66 AssignmentStatement()		

```
64      const requestObject = {
65        adapter: `${adapterEmergencyContact}/updateEmergencycontact`,
66        method: 'post',
67        parameters: { params: `['${agentNumber}','${contactName}','${relation}','${phoneNumber}','${address1}','${address2}','${address3}']` },
68      };
```

UserProfileService.js, line 88 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		

Abstract:	The HTTP request at UserProfileService.js line 88 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.
Sink:	UserProfileService.js:88 AssignmentStatement()

```
86    const requestObject = {
87      adapter: `${adapterEmergencyContact}/checkEmergencyContact`,
88      method: 'post',
89      parameters: { params: `['${agentCode}']` },
90    };
```

UserProfileService.js, line 110 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The HTTP request at UserProfileService.js line 110 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	UserProfileService.js:110 AssignmentStatement()		

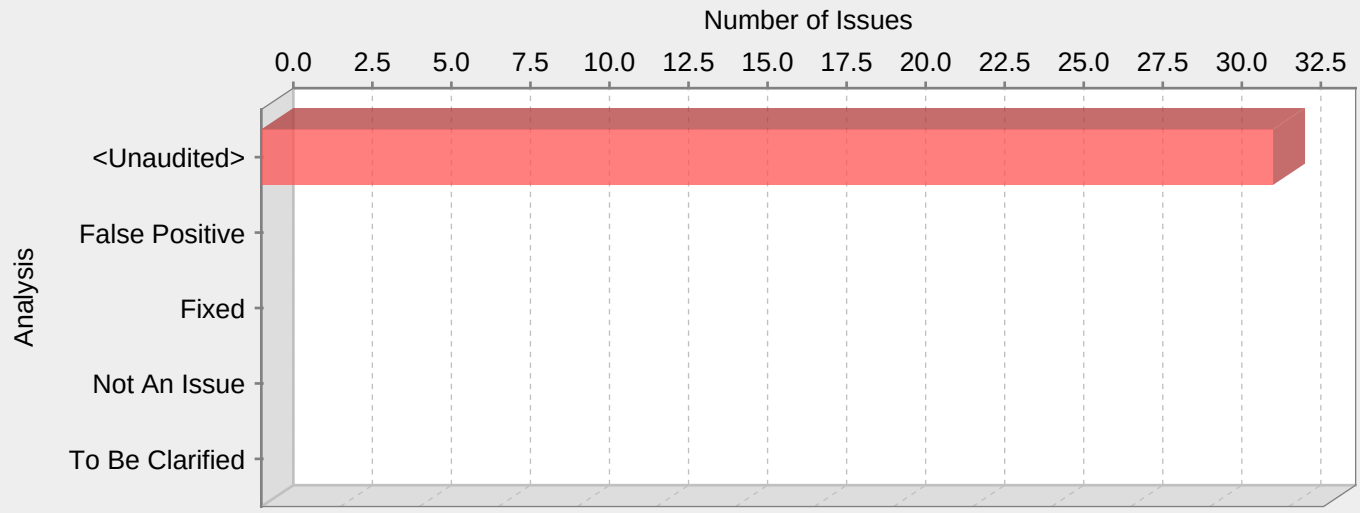
```
108   const requestObject = {
109     adapter: `${adapterEmergencyContact}/showEmergencyContact`,
110     method: 'post',
111     parameters: { params: `['${agentCode}']` },
112   };
```

UserProfileService.js, line 22 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The HTTP request at UserProfileService.js line 22 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	UserProfileService.js:22 AssignmentStatement()		

```
20    const requestObject = {
21      adapter: `${adapterEmergencyContact}/getRelation`,
22      method: 'post',
23      parameters: { params: "['']" },
24    };
```

Category: Weak Cryptographic Hash (32 Issues)



Abstract:

Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.

Explanation:

MD2, MD4, MD5, RIPEMD-160, and SHA-1 are popular cryptographic hash algorithms often used to verify the integrity of messages and other data. However, as recent cryptanalysis research has revealed fundamental weaknesses in these algorithms, they should no longer be used within security-critical contexts.

Effective techniques for breaking MD and RIPEMD hashes are widely available, so those algorithms should not be relied upon for security. In the case of SHA-1, current techniques still require a significant amount of computational power and are more difficult to implement. However, attackers have found the Achilles' heel for the algorithm, and techniques for breaking it will likely lead to the discovery of even faster attacks.

Recommendations:

Discontinue the use of MD2, MD4, MD5, RIPEMD-160, and SHA-1 for data-verification in security-critical contexts. Currently, SHA-224, SHA-256, SHA-384, SHA-512, and SHA-3 are good alternatives. However, these variants of the Secure Hash Algorithm have not been scrutinized as closely as SHA-1, so be mindful of future research that might impact the security of these algorithms.

MyAccountChangePassword.js, line 123 (Weak Cryptographic Hash)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	MyAccountChangePassword.js:123 FunctionPointerCall: SHA1()		

```
121     const oldPassword =****** '');
122     if (!isPublic) {
123       return this.props.auth.pwd !==
CryptoJS.SHA1(JSON.stringify(this.state.currentOldPassword)).toString();
124     }
125     return oldPassword !==
CryptoJS.SHA1(JSON.stringify(this.state.currentOldPassword)).toString();
```

SignInHome.js, line 97 (Weak Cryptographic Hash)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	SignInHome.js:97 FunctionPointerCall: SHA1()		

```
95                  isPublic: true,
96                  username: this.state.userID,
97                  oldPassword:
CryptoJS.SHA1(JSON.stringify(this.state.password)).toString(),
98                }),
99              }],
```

MyAccountChangePassword.js, line 102 (Weak Cryptographic Hash)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	MyAccountChangePassword.js:102 FunctionPointerCall: SHA1()		

```
100         if (this.props.apiStatus === requestStatus.FAILED) Alert.alert(_('Error'),
this.props.setting.changePasswordError);
101         if (this.props.apiStatus === requestStatus.SUCCESS) {
102
this.props.hashingUserPassword(CryptoJS.SHA1(JSON.stringify(this.state.newPassword)).t
oString());
103           Alert.alert('', 'Password berhasil diganti!', [{
104             text: 'OK',
```

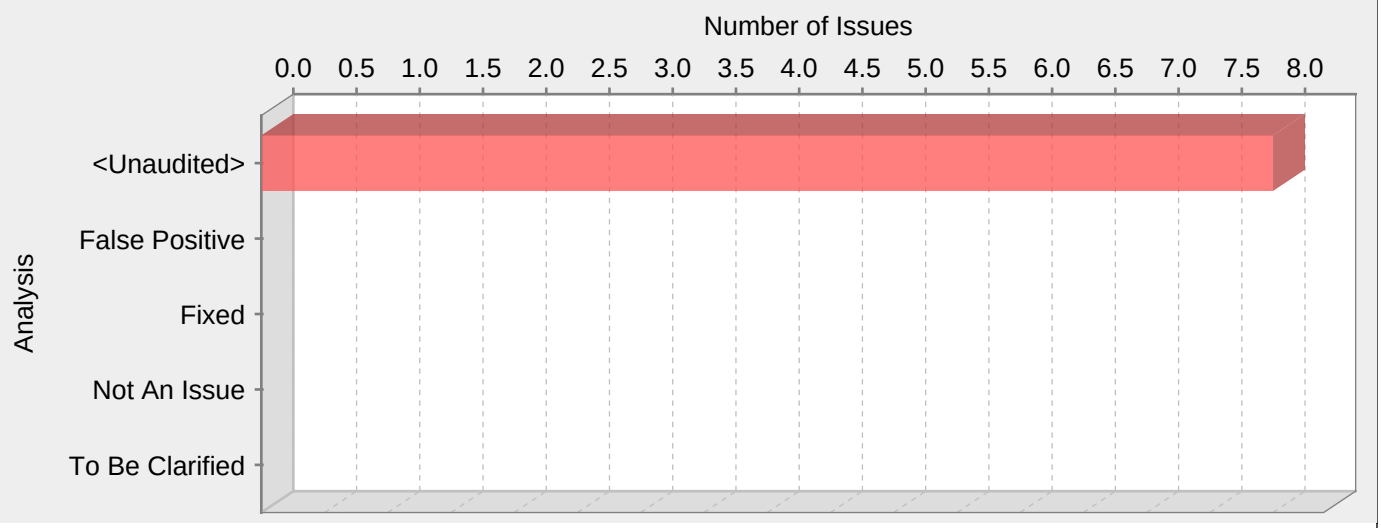
SignInHome.js, line 118 (Weak Cryptographic Hash)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	SignInHome.js:118 FunctionPointerCall: SHA1()		

```
116               ],
117               { cancelable: false });
118           } else if (this.props.onlineDate !== undefined &&
CryptoJS.SHA1(JSON.stringify(this.state.password)).toString() === this.props.pwd
119             && this.props.res && this.props.res.username &&
this.props.res.username.toLowerCase() === this.state.userID.toLowerCase()
120             && !(this.props.res.agent_channelType === '1' ||
this.props.res.agent_channelType === 1) && this.props.err.code === '999') {
```

MyAccountChangePassword.js, line 125 (Weak Cryptographic Hash)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	MyAccountChangePassword.js:125 FunctionPointerCall: SHA1()		

```
123       return this.props.auth.pwd !==
CryptoJS.SHA1(JSON.stringify(this.state.currentOldPassword)).toString();
124     }
125     return oldPassword !==
CryptoJS.SHA1(JSON.stringify(this.state.currentOldPassword)).toString();
126   }
```

Category: Password Management: Password in Comment (8 Issues)



Abstract:

Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.

Explanation:

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

Example: The following comment specifies the default password to connect to a database:

...
// Default username for database connection is "scott"
// Default password for database connection is "tiger"
...

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is likely no way to change the database user "scott" with a password of "tiger" unless the program is patched. An employee with access to this information could use it to break into the system.

Recommendations:

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

Tips:

1. Avoid hardcoding passwords in source code and avoid using default passwords. If a hardcoded password is the default, require that it be changed and remove it from the source code.

index.js, line 608 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		

Sink: index.js:608 Comment()

```
606  export default InputField;
607
608  /**
609  keyboardType =
```

ConfigDashboard.js, line 47 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		

Sink:	ConfigDashboard.js:47 Comment()
45	`export const THIRD_CAMPAIGN_FAILED = 'THIRD_CAMPAIGN_FAILED';`
46	
47	`// SETTINGS - CHANGE PASSWORD`
48	
49	`export const CHANGE_PASSWORD_API = 'adapters/HTTPAdapterAuth/ChangePasspruforceid';`

index.js, line 492 (Password Management: Password in Comment)

Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		

Sink:	index.js:492 Comment()
490	`export default InputField;`
491	
492	`/**`
493	`keyboardType =`

MyAccountChangePassword.js, line 34 (Password Management: Password in Comment)

Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		

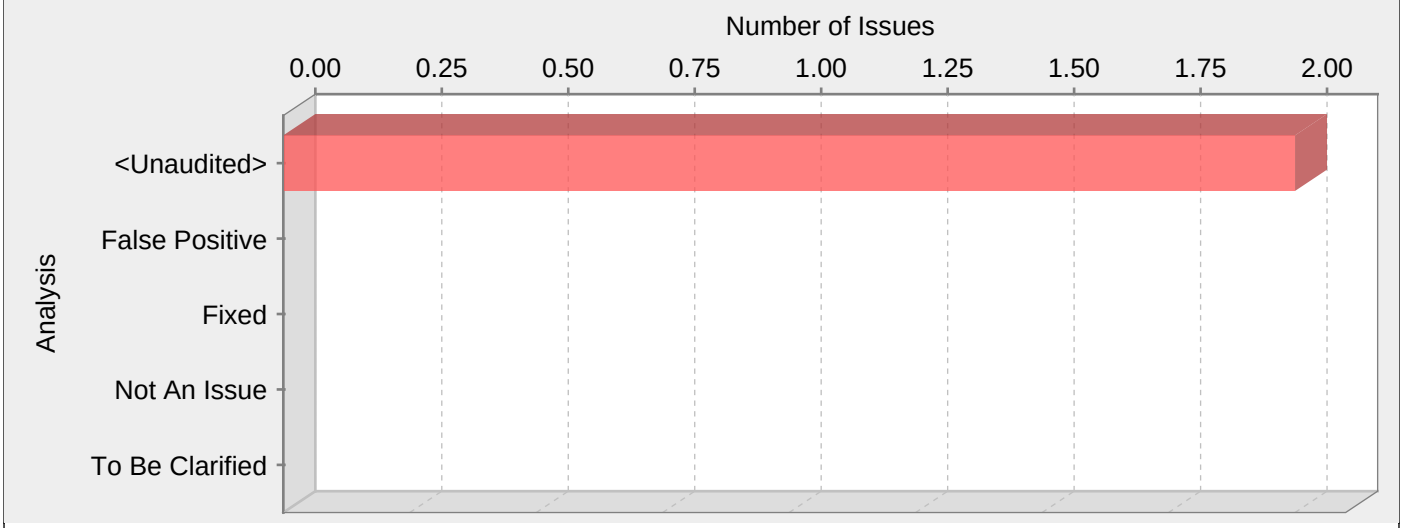
Sink:	MyAccountChangePassword.js:34 Comment()
32	`sandiLama: true,`
33	`currentOldPassword: '',`
34	`// oldPassword: 'A',`
35	`hideOldPassword: true,`
36	`oldPasswordIcon: 'eye-slash',`

SignInHome.js, line 45 (Password Management: Password in Comment)

Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plaintext anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		

Sink:	SignInHome.js:45 Comment()
43	`this.state = {`
44	`userID: this.props.navigation.getParam('agentData', {}).salesforceId || '',`
45	`// password: this.props.navigation.getParam('agentData', {}).password || '',`
46	`password: '',`
47	`showLoadingModal: false,`

Category: Password Management: Null Password (2 Issues)



Abstract:
Null passwords can lead to confusion in the code.

Explanation:
It is not a good idea to have a null password.

Example: The following code sets the password to initially to null:

```
...
var password=null;
...
{
password=getPassword(user_data);
...
}
...
if(password==null){
// Assumption that the get didn't work
...
}
...
```

Recommendations:
To avoid confusion, password variables should immediately be assigned to the correct variable.

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

ReducerAuth.js, line 65 (Password Management: Null Password)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can lead to confusion in the code.		
Sink:	ReducerAuth.js:65 FieldAccess: pwd()		

```
63    // },
64    err: null,
65    pwd: null,
66  };
```

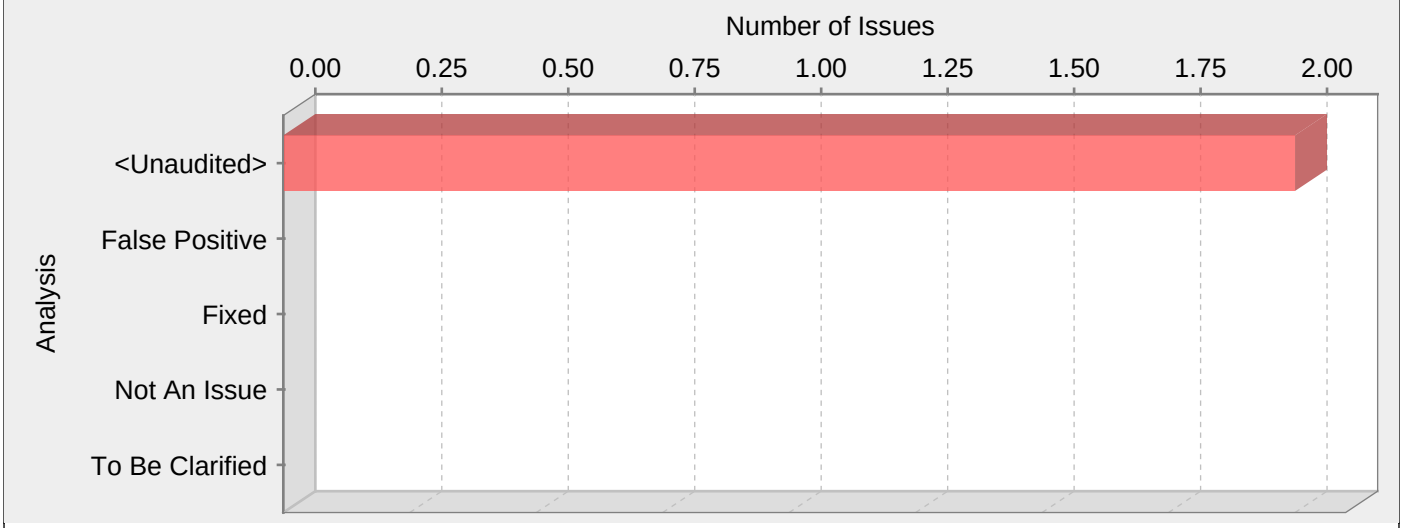
ReducerIncomeStatement.js, line 174 (Password Management: Null Password)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		

Abstract:	Null passwords can lead to confusion in the code.
Sink:	ReducerIncomeStatement.js:174 FieldAccess: pwd()

```
172    pdf: null,
173    email: null,
174    pwd: null,
175    sendEmail: null,
```

Category: SQL Injection (2 Issues)



Abstract:

On line 28 of SQLiteUtils.js, the program invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

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

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

```
...
var username = document.form.username.value;
var itemName = document.form.itemName.value;
var query = "SELECT * FROM items WHERE owner = " + username + " AND itemname = " + itemName + ";";
db.transaction(function (tx) {
tx.executeSql(query);
}
)
...
```

The query that this code intends to execute follows:

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

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

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

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

```
SELECT * FROM items;
```

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

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

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

DELETE FROM items;

--'
```

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

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

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

DELETE FROM items;

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

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

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

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

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

Recommendations:

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

When connecting to MySQL, the previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
...
var username = document.form.username.value;
var itemName = document.form.itemName.value;
var query = "SELECT * FROM items WHERE owner = ? AND itemname = ? ;";
db.transaction(function (tx) {
tx.executeSql(query,[username,itemName]);
}
)
```

...

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

Tips:

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

SQLiteUtils.js, line 33 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 33 of SQLiteUtils.js, the program invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Sink:	SQLiteUtils.js:33 FunctionPointerCall: executeSql()		

```
31  const executeTransaction = (db, query, params) => new Promise((resolve, reject) => {
32    db.transaction(
33      tx => tx.executeSql(query, params),
34      error => reject(error),
35      () => resolve('OK'),
```

SQLiteUtils.js, line 28 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 28 of SQLiteUtils.js, the program invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Sink:	SQLiteUtils.js:28 FunctionPointerCall: executeSql()		

```
26
27  const executeQuery = (db, query, params) => new Promise((resolve, reject) => {
28    db.executeSql(query, params, x => resolve(x), y => reject(y));
29  });
```