Fortify Developer Workbook

May 21, 2018

Report Overview

Report Summary

On May 21, 2018, a source code review was performed over the TASCore code base. 283 files, 24,155 LOC (Executable) were scanned. A total of 22 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	
Critical	13
Low	8
High	1

Issue Summary

Overall number of results

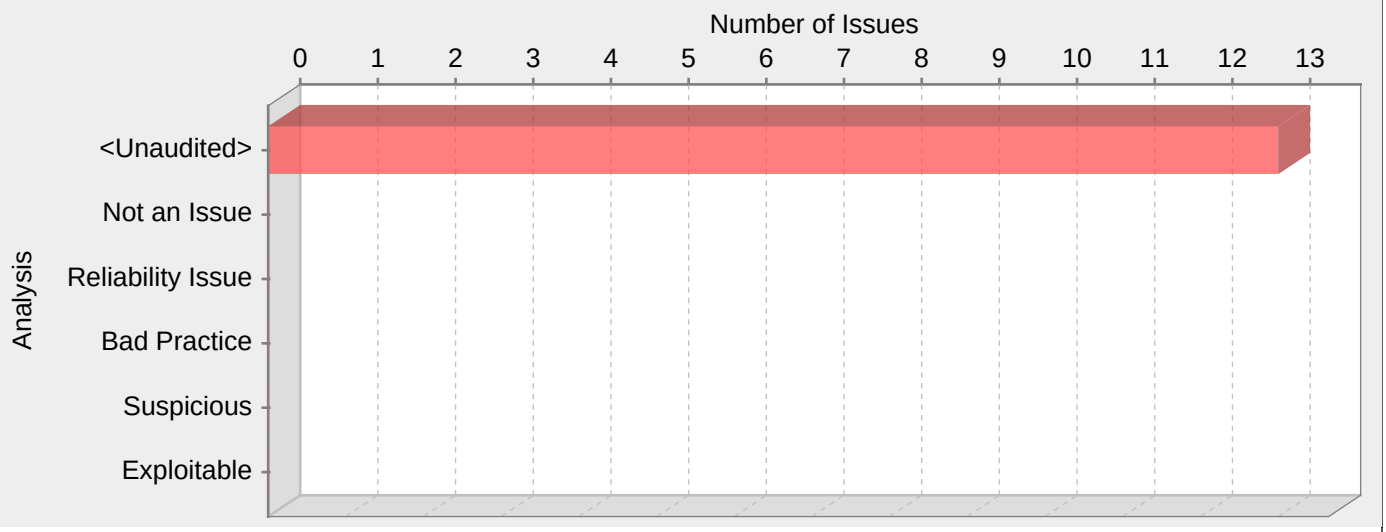
The scan found 22 issues.

Issues by Category	
Key Management: Hardcoded Encryption Key	13
HTML5: Overly Permissive Message Posting Policy	5
Cross-Site Request Forgery	2
Insecure Randomness	1
Password Management: Password in Comment	1

Results Outline

Vulnerability Examples by Category

Category: Key Management: Hardcoded Encryption Key (13 Issues)



Abstract:

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

Explanation:

It is never a good idea to hardcode an encryption key because it allows all of the project's developers to view the encryption key, and makes fixing the problem extremely difficult. Once the code is in production, the encryption key cannot be changed without patching the software. If the account that is protected by the encryption key is compromised, the owners of the system will be forced to choose between security and availability.

Example 1: The following code uses a hardcoded encryption key:

...
var crypto = require('crypto');
var encryptionKey = "lakdsljkalkjlksdfkl";
var algorithm = 'aes-256-ctr';
var cipher = crypto.createCipher(algorithm, encryptionKey);
...

Anyone who has access to the code will have access to the encryption key. Once the application has shipped, there is no way to change the encryption key unless the program is patched. An employee with access to this information could use it to break into the system. Even worse, if attackers had access to the executable for the application, they could extract the encryption key value.

Recommendations:

Encryption keys should never be hardcoded and should be obfuscated and managed in an external source. Storing encryption keys in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the encryption key.

tableau-2.2.2.js, line 7015 (Key Management: Hardcoded Encryption Key)

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

Sink: tableau-2.2.2.js:7015 Operation()

```
7013        while ($t1.moveNext()) {
7014          var entry = $t1.current();
7015          if (entry.key !== 'embed' && entry.key !== 'height' && entry.key !== 'width' &&
entry.key !== 'device' && entry.key !== 'autoSize' && entry.key !== 'hideTabs' &&
entry.key !== 'hideToolbar' && entry.key !== 'onFirstInteractive' && entry.key !==
'onFirstVizSizeKnown' && entry.key !== 'toolbarPosition' && entry.key !==
'instanceIdToClone' && entry.key !== 'navType' && entry.key !==
'display_static_image') {
7016            url.push('&');
7017            url.push(encodeURIComponent(entry.key));
```

tableau-2.2.2.js, line 7015 (Key Management: Hardcoded Encryption Key)

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

Sink: tableau-2.2.2.js:7015 Operation()

```
7013        while ($t1.moveNext()) {
7014          var entry = $t1.current();
7015          if (entry.key !== 'embed' && entry.key !== 'height' && entry.key !== 'width' &&
entry.key !== 'device' && entry.key !== 'autoSize' && entry.key !== 'hideTabs' &&
entry.key !== 'hideToolbar' && entry.key !== 'onFirstInteractive' && entry.key !==
'onFirstVizSizeKnown' && entry.key !== 'toolbarPosition' && entry.key !==
'instanceIdToClone' && entry.key !== 'navType' && entry.key !==
'display_static_image') {
7016          url.push('&');
7017          url.push(encodeURIComponent(entry.key));
```


tableau-2.2.2.js, line 7015 (Key Management: Hardcoded Encryption Key)

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

Sink: tableau-2.2.2.js:7015 Operation()

```
7013        while ($t1.moveNext()) {
7014          var entry = $t1.current();
7015          if (entry.key !== 'embed' && entry.key !== 'height' && entry.key !== 'width' &&
entry.key !== 'device' && entry.key !== 'autoSize' && entry.key !== 'hideTabs' &&
entry.key !== 'hideToolbar' && entry.key !== 'onFirstInteractive' && entry.key !==
'onFirstVizSizeKnown' && entry.key !== 'toolbarPosition' && entry.key !==
'instanceIdToClone' && entry.key !== 'navType' && entry.key !==
'display_static_image') {
7016          url.push('&');
7017          url.push(encodeURIComponent(entry.key));
```

tableau-2.2.2.js, line 7015 (Key Management: Hardcoded Encryption Key)

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

Sink: tableau-2.2.2.js:7015 Operation()

```
7013        while ($t1.moveNext()) {
7014          var entry = $t1.current();
7015          if (entry.key !== 'embed' && entry.key !== 'height' && entry.key !== 'width' &&
entry.key !== 'device' && entry.key !== 'autoSize' && entry.key !== 'hideTabs' &&
entry.key !== 'hideToolbar' && entry.key !== 'onFirstInteractive' && entry.key !==
'onFirstVizSizeKnown' && entry.key !== 'toolbarPosition' && entry.key !==
'instanceIdToClone' && entry.key !== 'navType' && entry.key !==
'display_static_image') {
7016          url.push('&');
7017          url.push(encodeURIComponent(entry.key));
```

tableau-2.2.2.js, line 7015 (Key Management: Hardcoded Encryption Key)

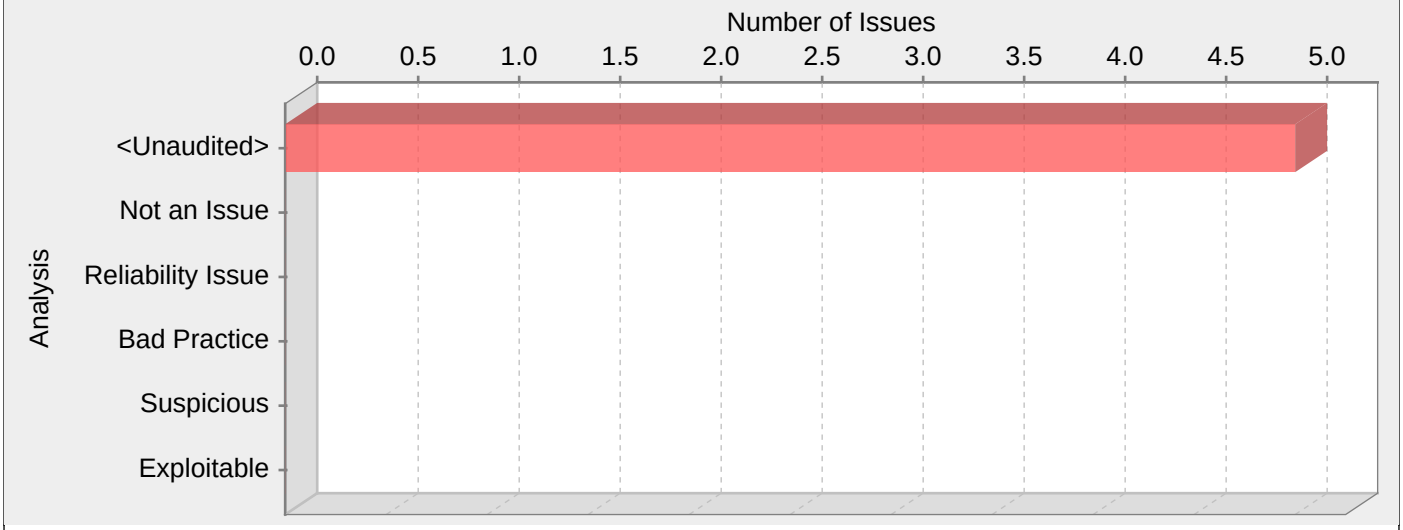
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Hardcoded encryption keys may compromise system security in a way that cannot be easily remedied.		

Sink: tableau-2.2.2.js:7015 Operation()

```
7013        while ($t1.moveNext()) {
7014          var entry = $t1.current();
7015          if (entry.key !== 'embed' && entry.key !== 'height' && entry.key !== 'width' &&
entry.key !== 'device' && entry.key !== 'autoSize' && entry.key !== 'hideTabs' &&
entry.key !== 'hideToolbar' && entry.key !== 'onFirstInteractive' && entry.key !==
'onFirstVizSizeKnown' && entry.key !== 'toolbarPosition' && entry.key !==
'instanceIdToClone' && entry.key !== 'navType' && entry.key !==
'display_static_image') {
```

```
7016                        url.push('&');
7017                        url.push(encodeURIComponent(entry.key));
```

Category: HTML5: Overly Permissive Message Posting Policy (5 Issues)



Abstract:

On line 4572 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..

Explanation:

One of the new features of HTML5 is cross-document messaging. The feature allows scripts to post messages to other windows. The corresponding API allows the user to specify the origin of the target window. However, caution should be taken when specifying the target origin because an overly permissive target origin will allow a malicious script to communicate with the victim window in an inappropriate way, leading to spoofing, data theft, relay and other attacks.

Example 1: Below is an example of using a wildcard to programmatically specify the target origin of the message to be sent.

```
o.contentWindow.postMessage(message, '*');
```

Using the * as the value of the target origin indicates that the script is sending a message to a window regardless of its origin.

Recommendations:

Do not use the * as the value of the target origin. Instead, provide a specific target origin.

Example 2: The code below provides a specific value for the target origin.

```
o.contentWindow.postMessage(message, 'www.trusted.com');
```

tableau-2.2.2.js, line 4572 (HTML5: Overly Permissive Message Posting Policy)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	On line 4572 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..		
Sink:	tableau-2.2.2.js:4572 FunctionPointerCall()		

```
4570        if ($tab__Utility.isPostMessageSynchronous()) {
4571         window.setTimeout(function() {
4572          iframe.contentWindow.postMessage(message, '*');
4573         }, 0);
4574        }
```

tableau-2.2.2.js, line 8626 (HTML5: Overly Permissive Message Posting Policy)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	On line 8626 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..		
Sink:	tableau-2.2.2.js:8626 FunctionPointerCall()		

```
8624          var command = new tab.NonApiCommand('sf', [requestId, scaleFactor.toString(),
scrollX.toString(), scrollY.toString()]);
8625          if (ss.isValue(this.$iframe) && ss.isValue(this.$iframe.contentWindow)) {
8626           this.$iframe.contentWindow.postMessage(command.serialize(), '*');
8627          }
8628         },
```

tableau-2.2.2.js, line 8612 (HTML5: Overly Permissive Message Posting Policy)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	On line 8612 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..		
Sink:	tableau-2.2.2.js:8612 FunctionPointerCall()		

```
8610                }
8611                var command = new tab.NonApiCommand('tableau.enableVisibleRectCommunication', []);
8612                this.$iframe.contentWindow.postMessage(command.serialize(), '*');
8613              },
8614              $redoAsync: function VizImpl$RedoAsync() {
```

tableau-2.2.2.js, line 4608 (HTML5: Overly Permissive Message Posting Policy)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	On line 4608 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..		
Sink:	tableau-2.2.2.js:4608 FunctionPointerCall()		

```
4606            if (command.get_name() === 'api.FirstVizSizeKnownEvent') {
4607              var bootstrapCommand = new $tab_NonApiCommand('tableau.bootstrap', []);
4608              messageEvent.source.postMessage(bootstrapCommand.serialize(), '*');
4609            }
4610          }
```

tableau-2.2.2.js, line 4576 (HTML5: Overly Permissive Message Posting Policy)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	On line 4576 of tableau-2.2.2.js the program posts a cross-document message with an overly permissive target origin..		
Sink:	tableau-2.2.2.js:4576 FunctionPointerCall()		

```
4574          }
4575          else {
4576            iframe.contentWindow.postMessage(message, '*');
4577          }
4578        };
```

Category: Cross-Site Request Forgery (2 Issues)

Abstract:

The form post at main-content.component.html line 10 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 by submitting this form:

```
<form method="POST" action="/new_user" >
Name of new user: <input type="text" name="username">
Password for new user: <input type="password" name="user_passwd">
<input type="submit" name="action" value="Create User">
</form>
```

An attacker might set up a Web site with the following:

```
<form method="POST" action="http://www.example.com/new_user">
<input type="hidden" name="username" value="hacker">
<input type="hidden" name="user_passwd" value="hacked">
</form>
<script>
document.usr_form.submit();
</script>
```

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 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.

main-content.component.html, line 10 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at main-content.component.html line 10 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	main-content.component.html:10 null()		

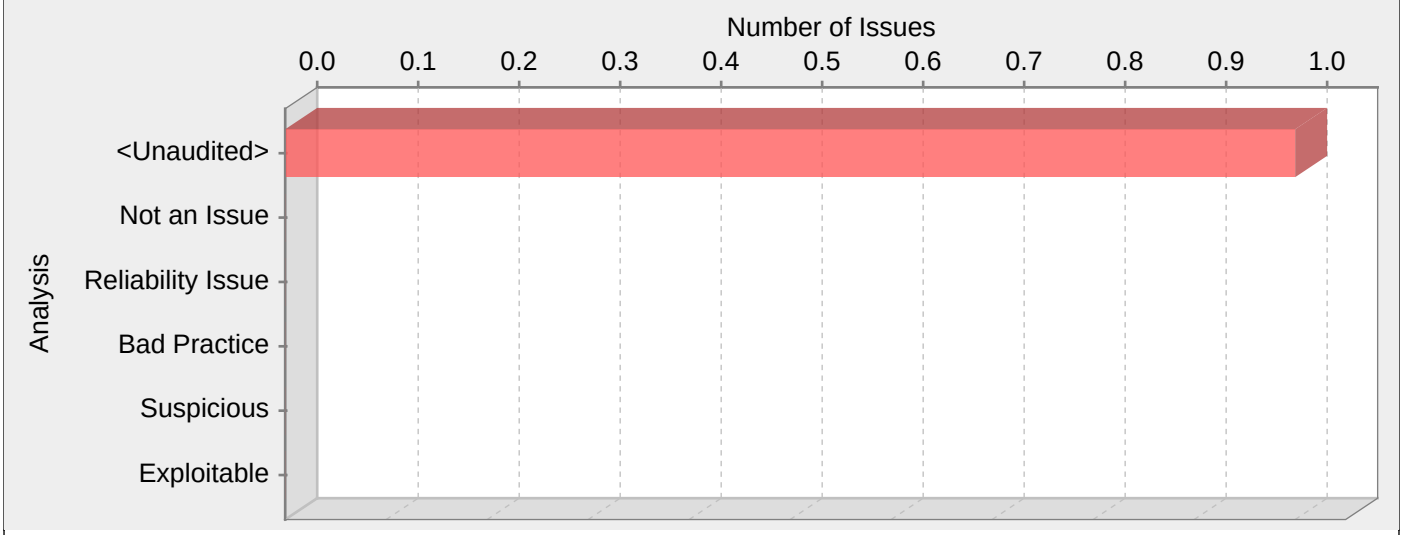
```
8
9                 <div *ngIf="mainContent">
10                  <form class="usa-form-small">
11                      <fieldset>
12                        <legend>App Content</legend>
```

reports-config.component.html, line 14 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at reports-config.component.html line 14 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	reports-config.component.html:14 null()		

```
12                  <app-tas-breadcrumb></app-tas-breadcrumb>
13
14                  <form class="usa-form-small" *ngIf="reportsConfig && !isSaving">
15                    <fieldset>
16                      <legend>Manage Reports Extract</legend>
```

Category: Insecure Randomness (1 Issues)



Abstract:

Standard pseudorandom number generators cannot withstand cryptographic attacks.

Explanation:

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

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

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

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

```
function genReceiptURL (baseURL){
var randNum = Math.random();
var receiptURL = baseURL + randNum + ".html";
return receiptURL;
}
```

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

Recommendations:

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

In JavaScript, the typical recommendation is to use the window.crypto.random() function in the Mozilla API. However, this method does not work in many browsers, including more recent versions of Mozilla Firefox. There is currently no cross-browser solution for a robust cryptographic PRNG. In the meantime, consider handling any PRNG functionality outside of JavaScript.

tableau-2.2.2.js, line 116 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Standard pseudorandom number generators cannot withstand cryptographic attacks.		
Sink:	tableau-2.2.2.js:116 FunctionPointerCall()		

```
114
115     ss.defaultHashCode = function ss$defaultHashCode(obj) {
```

```
116                   return obj.$__hashCode__ || (obj.$__hashCode__ = (Math.random() * 0x100000000) | 0);
117                 };
```

Category: Password Management: Password in Comment (1 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.

uswds.js, line 1771 (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: uswds.js:1771 Comment()

```
1769  };
1770
1771  /**
1772   * Component that decorates an HTML element with the ability to toggle the
1773   * masked state of an input field (like a password) when clicked.
```