Password hashing: Be careful about what you hash!

Implementing password storage for user authentication is difficult and error-prone. In case of implementation mistake, the password will be exposed to attacks such as password cracking. Let us show you, in this article, a real situation where even with the use of a recommended and strong hashing algorithm, the authentication functionality of a web application has collapsed.

January 18, 2021

Share

Context of the hashing issue

During a web assessment, Excellium's Intrusion & AppSec team audited a PHP application where users passwords were stored using the bcrypt hashing algorithm. As bcrypt [1] is still a valid and recommended algorithm to hash passwords, compromising passwords [2] should not be an easy task. However, sometimes, the devil is in the details.

Multiple shades of bcrypt

After discovering a boolean-based blind SQL injection[3], followed by a Remote Code Execution[4] vulnerability, the team could collect some user hashes and download the application source code. They analysed it and looked for other vulnerabilities from this new point of view.

The following code snippet is the hash computation of a password (*some parts were redacted to not disclose the real source code of the product*):

```
<?php
define("SALT", "SALT");
function modifyPassword($password){
 $email = 'XLM';
  return hash_hmac('whirlpool', str_pad($password, strlen($password)*4, sha1($email), STR_PAD_BOTH), SALT, true);
}
function computeHash($password){
 $modifiedPassword = modifyPassword($password);
  return password_hash($modifiedPassword, PASSWORD_BCRYPT, array('cost' => 4));
}
function verifyHash($password, $hash){
  $modifiedPassword = modifyPassword($password);
  return password_verify($modifiedPassword, $hash);
}
?>
```

The hashing function **computeHash()** is not strictly bcrypt algorithm applied to the password, but it alters the password (with **modifyPassword()** function) before hashing it.

With the collected user hashes on one hand and the code on the other, all pieces are gathered to attempt passwords compromise using an offline dictionary attack. To not disclose real hashes from the assessment (especially administrative users ones), the PHP script below starts by computing the hash of the password '*monkey12*'. It then performs the attack, by looping on the well-known **rockyou**[5] dictionary and print each entry that leads to this hash.

```
<?php
$initialPwd = "monkey12";
$h = computeHash($initialPwd);
echo("\n[+] Initial password:\n$initialPwd\n");
echo("[+] Generated hash:\n$h\n");
echo("[+] Search for matching password:\n");
$handle = fopen("rockyou.txt", "r");
while <span class="footnote_referrer"><a role="button" tabindex="0" onclick="footnote_moveToReference_75341_1('footnote_plugin_reference_75341_1_1');" onkeypress="footnote_moveToReference_75341_1('footnote_plugin_reference_75341_1_1');"><sup id="footnote_plugin_tooltip_75341_1_1" class="footnote_plugin_tooltip_text">[1]</sup></a><span id="footnote_plugin_tooltip_text_75341_1_1" class="footnote_tooltip">$password = fgets($handle</span></span><script type="text/javascript"> jQuery('#footnote_plugin_tooltip_75341_1_1').tooltip({ tip: '#footnote_plugin_tooltip_text_75341_1_1', tipClass: 'footnote_tooltip', effect: 'fade', predelay: 0, fadeInSpeed: 200, delay: 400, fadeOutSpeed: 200, position: 'top center', relative: true, offset: [-7, 0], });</script> !== false) {
  $pwd = trim($password);
  if($pwd === $initialPwd){
    continue;
   }   if(verifyHash($pwd, $h)){
    echo("$pwd\n");   }
 }
echo("\n[+] Search finished.");
fclose($handle); ?>
```

Here comes the surprise, during the execution of the script above, the following results were obtained after less than 15 minutes:

```
$ php code.php
[+] Initial password:
monkey12
[+] Generated hash:
$2y$04$we4stlFLrbAzEoGBP7G1.Ov59OOpKoCQGs7M0w0UsFlAxNzzJSEqG
[+] Search for matching password:
090987
brittani1
trouble12
110959
…
```

These 4 entries are all valid passwords, even if not equal to '*monkey12*'! Indeed, when submitting one of them in the application, the **verifyHash()** function will conclude that the stored hash matches with the hash computed from the provided password, and therefore lets an attacker log in.

As bcrypt is a valid and recommended hashing algorithm, this should not happen!

Indeed, as a cryptographic hash algorithm, bcrypt is resistant to second preimage attacks: it is computationally infeasible to find a message that yields a given hash value. This is at odds with the results above: we could quickly find 4 other words that yields to the hash of '*monkey12*'.

For the whole **rockyou** dictionary (14.344.383 entries), 233 matching passwords were found for the hash above: there is definitely something wrong with the **computeHash()** function outside of bcrypt algorithm. The team moved forward to identify the root cause of this behaviour.

Analysis of the behaviour

First of all, to not be fooled by a potential marginal case, the team ran again the script, changing the inputs and inspecting each intermediate results. Unfortunately (for the application), the unexpected and unwanted behaviour was also repeated.

As presented earlier, when validating a user password, the application calls the function **computeHash()**, that calls the function **modifyPassword()** to alter the initial password prior hashing it using the PHP built-in function **password_hash()**, which will use the **bcrypt** algorithm.

Also, in **modifyPassword()** function, a special flag ("*true*", in blue in the code snippet below) is passed to the PHP built-in function **hash_hmac()**: this flag indicates the value returned is **binary data**:

```
<?php
…
function modifyPassword($password){
 $email = 'XLM';
  return hash_hmac('whirlpool', str_pad($password, strlen($password)*4,
                sha1($email), STR_PAD_BOTH), SALT, true);
}
…
?>
```

So the input of **password_verify()**, the variable *$modifiedPassword*, is binary data. Despite no warning on the official documentation of **password_verify()**[6], this is a very bad idea in this specific case.

For non PHP developer, **password_verify()** function is used in conjunction with **password_hash()**: the later one create the hash from a password, with all information regarding the algorithm used in the result, while the former one decides if a provided password matches the hash, and therefore the initial password:

1. From algorithm information stored in the hash, it derives a hash from the provided password,
2. Then it compares the two hashes.

However, neither function was written to be "binary data safe" [7] when it hits the character **\0** (null byte, end of the string) in a password, it assumes the end of the data. In other words, a password with a null byte before the end will simply be truncated, and the hash will be therefore computed on a shorter value.

Consequently, the application does not hash the salted password directly but the *$modifiedPassword*, a binary data which is the result of the HMAC function used for salt injection. This other cryptographic function also uses a hashing algorithm internally, which means that there is a good probability to observe **\0** in the result.

Consequently, if a null byte arises soon in the variable *$modifiedPassword*, a lot of alternative passwords would succeed. For instance, if *$modifiedPassword* starts with **\0**, any other words that causes the HMAC to starts with **\0** would validate.

In the execution of the script above, a hash is generated for the password **monkey12** and the hash is valid for the password **090987**. Indeed, $modifiedPassword variable starts with **0xfc00** for both passwords, resulting in **password_verify()** only comparing the first character of the hash:

```
<?php
$pass = array("monkey12", "090987", "brittani1", "trouble12",  "110959");
foreach ($pass as &$value) {
  echo(bin2hex(modifyPassword($value)) . "\n");
}
?>
```

```
$ php code.php
fc00f1b5ba63d892ba314bed9b0d5529c3d80eb7d118454b8324088c…
fc004ee0eb4691e29b2c54069a9644cb3ccd3df72deff61d754b005d…
fc006a25169bb138cf6417c444838cacfef01c48be14ffc9564a2e7d…
fc0065cdfd744126d845fe20ffb1cfddbb8538b7d968dc0abbb53d49…
fc004693a552365ba5b3a19b8ae54bb039cb8d821e4c3ebe4eccf09c…
```

Consequence

The direct consequence is that in case of password guessing on an account then the chance to pass the authentication check is higher due to "collisions". This would be especially damageable for an administrative account.

Pinpoint and prevent the issue

The prevent the issue while keeping the hash hmac function in injecting the salt, set the last flag of the PHP built-in function **hash_hmac()** to **false** in order to indicate to return the HMAC encoded in hexadecimal.

More generally, do pay attention to the type of variables (boolean, integer, binary, string, …). Especially when the programming language is permissive like PHP.

A better alternative: keep it simple

Do not do pre-hashing or pre-processing to include a custom salt if the hashing algorithm used (like bcrypt) can generate a dedicated salt itself when hashing the password.

It is possible to include a common pepper [8] to each password to make offline password cracking operations harder without prior access to the source code or configuration. However, prefer string concatenation to update the password prior to passing it to the bcrypt computation function to avoid the kind of issue mentioned above.

Regarding the limit of bcrypt to 72 characters [9], just ensure that a user provides a strong password up to 70 characters. This size is already a strong password length, combined with bcrypt protection, against password cracking attacks.

Notification

The vulnerability was raised to the vendor, by Excellium's CSIRT [10], during the assessment.

Credits:

- Guenaëlle DE-JULIS
- Alexis PAIN
- Julien EHRHART
- Dominique RIGHETTO

References

↑1 $password = fgets($handle

Advice, General | Bcript, Hash, Password Hashing

Let's deliver the right solution for your business

CONTACT US