

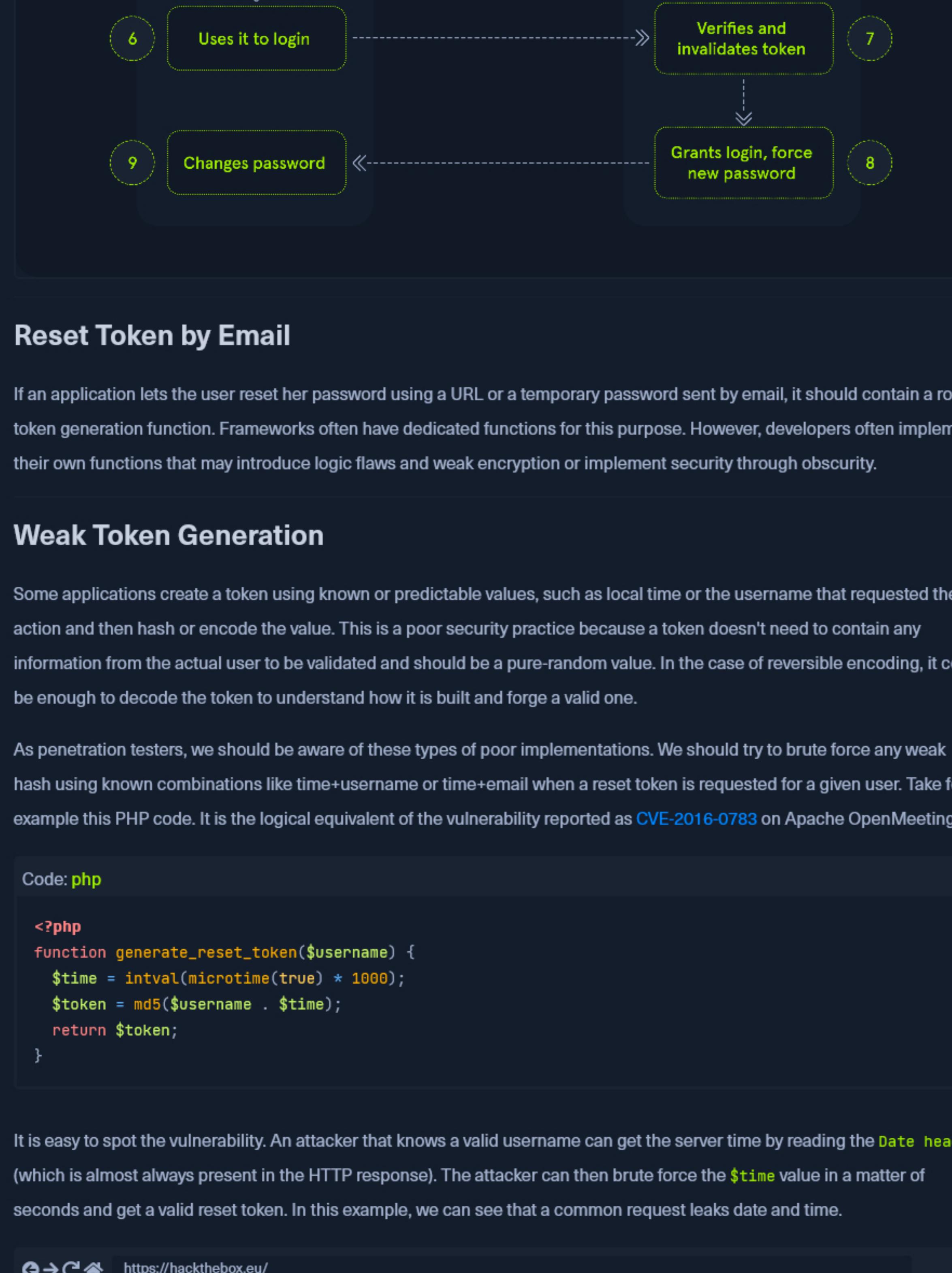
Predictable Reset Token

Reset tokens (in the form of a code or temporary password) are secret pieces of data generated mainly by the application when a password reset is requested. A user must provide it to prove their identity before actually changing their credentials.

Sometimes applications require you to choose one or more security questions and provide an answer at the time of registration. If you forgot your password, you could reset it by answering these questions again. We can consider these answers as tokens too.

This function allows us to reset the actual password of the user without knowing the password. There are several ways this can be done, which we will discuss soon.

A password reset flow may seem complicated since it usually consists of several steps that we must understand. Below, we created a basic flow that recaps what happens when a user requests a reset and receives a token by email. Some steps could go wrong, and a process that looks safe can be vulnerable.



Reset Token by Email

If an application lets the user reset her password using a URL or a temporary password sent by email, it should contain a robust token generation function. Frameworks often have dedicated functions for this purpose. However, developers often implement their own functions that may introduce logic flaws and weak encryption or implement security through obscurity.

Weak Token Generation

Some applications create a token using known or predictable values, such as local time or the username that requested the action and then hash or encode the value. This is a poor security practice because a token doesn't need to contain any information from the actual user to be validated and should be a pure-random value. In the case of reversible encoding, it could be enough to decode the token to understand how it is built and forge a valid one.

As penetration testers, we should be aware of these types of poor implementations. We should try to brute force any weak hash using known combinations like time+username or time+email when a reset token is requested for a given user. Take for example this PHP code. It is the logical equivalent of the vulnerability reported as [CVE-2016-0783](#) on Apache OpenMeeting:

Code: php

```
<?php
function generate_reset_token($username) {
    $time = intval(microtime(true) * 1000);
    $token = md5($username . $time);
    return $token;
}
```

It is easy to spot the vulnerability. An attacker that knows a valid username can get the server time by reading the `Date` header (which is almost always present in the HTTP response). The attacker can then brute force the `$time` value in a matter of seconds and get a valid reset token. In this example, we can see that a common request leaks date and time.

Let's take as an example the PHP code downloadable [here](#). The application generates a token by creating an md5 hash of the number of seconds since epoch (for demonstration purposes, we just use a time value). Reading the code, we can easily spot a vulnerability similar to the OpenMeeting one. Using the `reset_token_time.py` script, we could gain some confidence in creating and brute-forcing a time-based token. Download both scripts and try to get the welcome message.

Please bear in mind that any header could be stripped or altered by placing a reverse proxy in front of the application. However, we often have the chance to infer time in different ways. These are the time of a sent or received in-app message, an email header, or last login time, to name a few. Some applications do not check for the token age, giving an attacker plenty of time for a brute force attack. It has also been observed that some applications never invalidate or expire tokens, even if the token has been used. Retaining such a critical component active is quite risky since an attacker could find an old token and use it.

Short Tokens

Another bad practice is the use of short tokens. Probably to help mobile users, an application might generate a token with a length of 5/6 numerical characters that sometimes could be easily brute-forced. In reality, there is no need to use a short one because tokens are received mainly by e-mail and could be embedded in an HTTP link that can be validated using a simple GET call like https://127.0.0.1/reset.php?token=any_random_sequence. A token could, therefore, easily be a sequence of 32 characters, for example. Let us consider an application that generates tokens consisting of five digits for the sake of simplicity. Valid token values range from 00000 to 99999. At a rate of 10 checks per second, an attacker can brute force the entire range in about 3 hours.

Also, consider that the same application replies with a `Valid token` message if the submitted token is valid; otherwise, an `Invalid token` message is returned. If we wanted to perform a brute force attack against the abovementioned application's tokens, we could use `wfuzz`. Specifically, we could use a string match for the case-sensitive string `Valid` (`-ss "Valid"`). Of course, if we did not know how the web application replies when a valid token is submitted, we could use a "reverse match" by looking for any response that does not contain `Invalid token` using `-H "Invalid"`. Finally, a five-digit integer range can be specified and created in `wfuzz` using `-z range,00000-99999`. You can see the entire `wfuzz` command below.

```
Govardhan Gujji22@htb[~/htb]$ wfuzz -z range,00000-99999 --ss "Valid" "https://brokenauthentication.hackthebox.eu/token.php?user=admin&token=FUZZ"
=====
* Wfuzz 3.1.0 - The Web Fuzzer
=====
Target: https://brokenauthentication.hackthebox.eu/token.php?user=admin&token=FUZZ
Total requests: 100000
=====
ID      Response Lines Word Chars Payload
=====
0001112: 200     0 L   5 W   26 Ch   "1111"
00017665: 200    0 L   5 W   28 Ch   "17664"
^C
Finishing pending requests...
```

An attacker could obtain access as a user before the morning coffee by executing the above brute force attack at night. Both the user and a sysadmin that checks logs and network traffic will most probably notice an anomaly, but it could be too late. This edge case may sound unrealistic, but you will be surprised by the lack of security measures in the wild. Always try to brute force tokens during your tests, considering that such an attack is loud and can also cause a Denial of Service, so it should be executed with great care and possibly only after conferring with your client.

Weak Cryptography

Even cryptographically generated tokens could be predictable. It has been observed that some developers try to create their own crypto routine, often resorting to security through obscurity processes. Both cases usually lead to weak token randomness. Also, some cryptographic functions have proven to be less secure. Rolling your own encryption is never a good idea. To stay on the safe side, we should always use modern and well-known encryption algorithms that have been heavily reviewed. A fascinating use case on attacks against weak cryptography is the research performed by F-Secure lab on OpenCart, published [here](#).

Researchers discovered that the application uses the `mt_rand()` PHP function, which is known to be [vulnerable](#) due to lack of sufficient entropy during the random value generation process. OpenCart uses this vulnerable function to generate all random values, from CAPTCHA to session_id to reset tokens. Having access to some cryptographically insecure tokens makes it possible to identify the seed, leading to predicting any past and future token.

Attacking `mt_rand()` is not an easy task by any means, but proof of concept attacks have been released [here](#) and [here](#). `mt_rand()` should be therefore used with caution and taking into account the security implications. The OpenCart example was a serious case since an attacker could easily obtain some values generated using `mt_rand()` through CAPTCHA without even needing a valid user account.

Reset Token as Temp Password

It should be noted that some applications use reset tokens as actual temporary passwords. By design, any temporary password should be invalidated as soon as the user logs in and changes it. It is improbable that such temporary passwords are not invalidated immediately after use. That being said, try to be as thorough as possible and check if any reset tokens being used as temporary passwords can be reused.

There are higher chances that temporary passwords are being generated using a predictable algorithm like `mt_rand()`, `md5(username)`, etc., so make sure you test the algorithm's security by analyzing some captured tokens.

