Validate an E-Mail Address with PHP, the Right Way

Douglas Lovell

Abstract

Develop a working PHP function to validate e-mail addresses.

The Internet Engineering Task Force (IETF) document, RFC 3696, “Application Techniques for Checking and Transformation of Names” by John Klensin, gives several valid e-mail addresses that are rejected by many PHP validation routines. The addresses: Abc\@def@example.com, customer/department=shipping@example.com and !def!xyz%abc@example.com are all valid. One of the more popular regular expressions found in the literature rejects all of them:

"^[_.a-z0-9-]+(\.[_.a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*\([^\(\).]*\[a-z]{2,3}\)$"

This regular expression allows only the underscore (_) and hyphen (-) characters, numbers and lowercase alphabetic characters. Even assuming a preprocessing step that converts uppercase alphabetic characters to lowercase, the expression rejects addresses with valid characters, such as the slash (/), equal sign (=), exclamation point (!) and percent (%). The expression also requires that the highest-level domain component has only two or three characters, thus rejecting valid domains, such as .museum.

Another favorite regular expression solution is the following:

"^[a-zA-Z0-9_-.]+@[a-zA-Z0-9-.]+$"

This regular expression rejects all the valid examples in the preceding paragraph. It does have the grace to allow uppercase alphabetic characters, and it doesn't make the error of assuming a high-level domain name has only two or three characters. It allows invalid domain names, such as example..com.

Listing 1 shows an example from PHP Dev Shed (http://www.devshed.com/c/a/PHP/Email-Address-Verification-with-PHP/2). The code contains (at least) three errors. First, it fails to recognize many valid e-mail address characters, such as percent (%). Second, it splits the e-mail address into user name and domain parts at the at sign (@). E-mail addresses that contain a quoted at sign (@) will break this code. Third, it fails to check for host address DNS records. Hosts with a type A DNS entry will accept e-mail and may not necessarily publish a type MX entry. I'm not picking on the author at PHP Dev Shed. More than 100 reviewers gave this a four-out-of-five-star rating.

Listing 1. An Incorrect E-mail Validation

```
function checkEmail($email) {
    if(preg_match("/^[a-zA-Z0-9-]+([a-zA-Z0-9_\.-])+$/", $email)){
```
list($username,$domain)=split('@',$email);
if(!checkdnsrr($domain,'MX')) {
    return false;
}
return true;

One of the better solutions comes from Dave Child's blog at I Love Jack Daniel's (http://ilovejackdaniels.com), shown in Listing 2 (http://www.ilovejackdaniels.com/php/email-address-validation). Not only does Dave love good-old American whiskey, he also did some homework, read RFC 2822 and recognized the true range of characters valid in an e-mail user name. About 50 people have commented on this solution at the site, including a few corrections that have been incorporated into the original solution. The only major flaw in the code collectively developed at I Love Jack Daniel's is that it fails to allow for quoted characters, such as @, in the user name. It will reject an address with more than one at sign, so that it does not get tripped up splitting the user name and domain parts using explode("@", $email). A subjective criticism is that the code expends a lot of effort checking the length of each component of the domain portion—effort better spent simply trying a domain lookup. Others might appreciate the due diligence paid to checking the domain before executing a DNS lookup on the network.

Listing 2. A Better Example from ILoveJackDaniel's9585l2.qrk

function check_email_address($email) {
    // First, we check that there's one @ symbol,
    // and that the lengths are right.
    if (!ereg("^[^@]{1,64}@[^@]{1,255}$", $email)) {
        // Email invalid because wrong number of characters
        // in one section or wrong number of @ symbols.
        return false;
    }
    // Split it into sections to make life easier
    $email_array = explode("@", $email);
    $local_array = explode(".", $email_array[0]);
    for ($i = 0; $i < sizeof($local_array); $i++) {
        if (!ereg("^[A-Za-z0-9!#$%&'*+/=?^_`{|}~\-\.]\[A-Za-z0-9!#$%&'*+/=?^_`{|}~\-\]([\[A-Za-z0-9\[A-Za-z0-9-]([0-9])|("^[\[A-Za-z0-9-]\[\[A-Za-z0-9\]0,63])|("^[\[A-Za-z0-9-]\[\[A-Za-z0-9\]0,62])|$",
            $local_array[$i])) {
            return false;
        }
    }
    // Check if domain is IP. If not,
    // it should be valid domain name
    if (!ereg("^[\[0-9.]+\]?\$", $email_array[1])) {
        $domain_array = explode(".", $email_array[1]);
        if (sizeof($domain_array) < 2) {
            return false; // Not enough parts to domain
        }
        for ($i = 0; $i < sizeof($domain_array); $i++) {
            if (!ereg("^[A-Za-z0-9-]([0,61][A-Za-z0-9-])|"([A-Za-z0-9-]+)\$",
                $domain_array[$i])) {
                return false;
            }
        }
    }
    return true;
}
Requirements

IETF documents, RFC 1035 “Domain Implementation and Specification”, RFC 2234 “ABNF for Syntax Specifications”, RFC 2821 “Simple Mail Transfer Protocol”, RFC 2822 “Internet Message Format”, in addition to RFC 3696 (referenced earlier), all contain information relevant to e-mail address validation. RFC 2822 supersedes RFC 822 “Standard for ARPA Internet Text Messages” and makes it obsolete.

Following are the requirements for an e-mail address, with relevant references:

1. An e-mail address consists of local part and domain separated by an at sign (@) character (RFC 2822 3.4.1).
2. The local part may consist of alphabetic and numeric characters, and the following characters: !, #, $, %, &, ', *, +, -, /, =, ?, ^, _, `, {, |, } and ~, possibly with dot separators (.), inside, but not at the start, end or next to another dot separator (RFC 2822 3.2.4).
3. The local part may consist of a quoted string—that is, anything within quotes ("), including spaces (RFC 2822 3.2.5).
4. Quoted pairs (such as \@) are valid components of a local part, though an obsolete form from RFC 822 (RFC 2822 4.4).
5. The maximum length of a local part is 64 characters (RFC 2821 4.5.3.1).
6. A domain consists of labels separated by dot separators (RFC1035 2.3.1).
7. Domain labels start with an alphabetic character followed by zero or more alphabetic characters, numeric characters or the hyphen (-), ending with an alphabetic or numeric character (RFC 1035 2.3.1).
8. The maximum length of a label is 63 characters (RFC 1035 2.3.1).
9. The maximum length of a domain is 255 characters (RFC 2821 4.5.3.1).
10. The domain must be fully qualified and resolvable to a type A or type MX DNS address record (RFC 2821 3.6).

Requirement number four covers a now obsolete form that is arguably permissive. Agents issuing new addresses could legitimately disallow it; however, an existing address that uses this form remains a valid address.

The standard assumes a seven-bit character encoding, not multibyte characters. Consequently, according to RFC 2234, “alphabetic” corresponds to the Latin alphabet character ranges a–z and A–Z. Likewise, “numeric” refers to the digits 0–9. The lovely international standard Unicode alphabets are not accommodated—not even encoded as UTF-8. ASCII still rules here.

Developing a Better E-mail Validator

That's a lot of requirements! Most of them refer to the local part and domain. It makes sense, then, to
start with splitting the e-mail address around the at sign separator. Requirements 2–5 apply to the local part, and 6–10 apply to the domain.

The at sign can be escaped in the local name. Examples are, Abc\@def@example.com and "Abc@def"@example.com. This means an explode on the at sign, $split = explode("@", $email); or another similar trick to separate the local and domain parts will not always work. We can try removing escaped at signs, $cleanat = str_replace("\\@", "");, but that will miss pathological cases, such as Abc\@example.com. Fortunately, such escaped at signs are not allowed in the domain part. The last occurrence of the at sign must definitely be the separator. The way to separate the local and domain parts, then, is to use the strrpos function to find the last at sign in the e-mail string.

Listing 3 provides a better method for splitting the local part and domain of an e-mail address. The return type of strrpos will be boolean-valued false if the at sign does not occur in the e-mail string.

Listing 3. Splitting the Local Part and Domain

```php
isValid = true;
 atIndex = strrpos($email, "@");
if (is_bool($atIndex) && !$atIndex)
{
    isValid = false;
} else
{
    $domain = substr($email, $atIndex+1);
    $local = substr($email, 0, $atIndex);
    // ... work with domain and local parts
}
```

Let's start with the easy stuff. Checking the lengths of the local part and domain is simple. If those tests fail, there's no need to do the more complicated tests. Listing 4 shows the code for making the length tests.

Listing 4. Length Tests for Local Part and Domain

```php
$localLen = strlen($local);
$domainLen = strlen($domain);
if ($localLen < 1 || $localLen > 64)
{
    // local part length exceeded
    $isValid = false;
} else if ($domainLen < 1 || $domainLen > 255)
{
    // domain part length exceeded
    $isValid = false;
}
```

Now, the local part has one of two forms. It may have a begin and end quote with no unescaped embedded quotes. The local part, Doug "Ace" L. is an example. The second form for the local part is, (a+(\a+)*), where a stands for a whole slew of allowable characters. The second form is more common than the first; so, check for that first. Look for the quoted form after failing the unquoted form.
Characters quoted using the back slash (\@) pose a problem. This form allows doubling the back-slash character to get a back-slash character in the interpreted result (\\). This means we need to check for an odd number of back-slash characters quoting a non-back-slash character. We need to allow \\@ and reject \\\@.

It is possible to write a regular expression that finds an odd number of back slashes before a non-back-slash character. It is possible, but not pretty. The appeal is further reduced by the fact that the back-slash character is an escape character in PHP strings and an escape character in regular expressions. We need to write four back-slash characters in the PHP string representing the regular expression to show the regular expression interpreter a single back slash.

A more appealing solution is simply to strip all pairs of back-slash characters from the test string before checking it with the regular expression. The str_replace function fits the bill. Listing 5 shows a test for the content of the local part. Garrick, small font in Listing 5.

Listing 5. Partial Test for Valid Local Part Content9585l5.qrk

```php
if (!preg_match('/^(\\.\|[A-Za-z0-9!#%\&\_\=\=\\/\$\'\*\+\?\^\{\}\\~\.-])+$/',
    str_replace("\\\","",$local)))
{
    // character not valid in local part unless
    // local part is quoted
    if (!preg_match('/^"([^"\"]\|^\"])+$/',
        str_replace("\\\","",$local)))
    {
        $isValid = false;
    }
}
```

The regular expression in the outer test looks for a sequence of allowable or escaped characters. Failing that, the inner test looks for a sequence of escaped quote characters or any other character within a pair of quotes.

If you are validating an e-mail address entered as POST data, which is likely, you have to be careful about input that contains back-slash (\), single-quote (') or double-quote characters ("). PHP may or may not escape those characters with an extra back-slash character wherever they occur in POST data. The name for this behavior is magic_quotes_gpc, where gpc stands for get, post, cookie. You can have your code call the function, get_magic_quotes_gpc(), and strip the added slashes on an affirmative response. You also can ensure that the PHP.ini file disables this “feature”. Two other settings to watch for are magic_quotes_runtime and magic_quotes_sybase.

The two regular expressions in Listing 5 are appealing because they are relatively easy to comprehend and don't require repetition of the allowable character group, [A-Za-z0-9!#%\&\_\=\=/\$\'\*\+\?\^\{\}\\~\.-]. Here's a test for you. Why does the character group require two back-slash characters before the forward slash and one back-slash character before the single quote?

One deficiency of the outer test of Listing 5 is that it passes local part strings that include dots anywhere in the string. Requirement number two states that dots can't start or end the local part, and they can't appear together two or more times. We could address this by expanding the outer regular expression into form ^a+(\a+)+$, where a is (\\[A-Za-z0-9!#%&\_\=\=/\$\*\+\?\{\}\\~\.-]). We could, but that leads to a long, hard-to-read, repetitive expression that's difficult to believe in. It's clearer to add the simple checks.
shown in Listing 6.

**Listing 6. Check for dot placement in the local part.**

```php
if ($local[0] == '.' || $local[$localLen-1] == '.')
{
    // local part starts or ends with '.
    $isValid = false;
}
else if (preg_match('/\.\./', $local))
{
    // local part has two consecutive dots
    $isValid = false;
}
```

The local part is a wrap. The code now checks all local part requirements. Checking the domain will complete the e-mail validation. The code could check all of the labels in the domain separately, as does the whiskey-loving code shown in Listing 2, but, as hinted earlier, the solution presented here allows the DNS check to do most of the domain validation work.

Listing 7 makes a cursory check to ensure only valid characters in the domain part, with no repeated dots. It goes on to make DNS lookups for MX and A records. It makes the check for the A record only if the MX record check fails. The code in Listing 4 verified the length of the domain value. 

**Listing 7. Domain Checks**

```php
if (!preg_match('/^[A-Za-z0-9\-\.]+$/', $domain))
{
    // character not valid in domain part
    $isValid = false;
}
else if (preg_match('/\.\./', $domain))
{
    // domain part has two consecutive dots
    $isValid = false;
}
else if (!(checkdnsrr($domain,"MX") || checkdnsrr($domain, "A")))
{
    // domain not found in DNS
    $isValid = false;
}
```

So, is it good? You decide. But, it would be nice to test the logic to ensure that it at least is correct. Listing 8 contains a series of e-mail address test cases that any e-mail validation should pass.

**Listing 8. Test the e-mail validation function.**

```php
<?php
require("validEmail.php"); // your favorite here

function testEmail($email)
{
    echo $email;
    $pass = validEmail($email);
    if ($pass)
    {
        echo "Passed!
```

```php
```
Be sure to run the test to see the valid and rejected e-mail addresses, the double-escaping (\) inside the PHP strings tends to obfuscate the addresses. You're challenged to subject your favorite e-mail validation code to this test. Be assured that the code in Listing 9 does pass!

Listing 9 contains a complete function for validating an e-mail address. It isn't as concise as many—it certainly isn't a one-liner. But, it is straightforward to read and comprehend, and it correctly accepts and rejects e-mail addresses that many other published functions incorrectly reject and accept. The function orders the validation tests roughly according to increasing cost. In particular, the more complex regular expression and, certainly, the DNS lookup, both come last.
Listing 9. A Complete E-mail Validation Function

```php
/**
 * Validate an email address.
 * Provide email address (raw input)
 * Returns true if the email address has the email
texture format and the domain exists.
 */
function validEmail($email) {
    $isValid = true;
    $atIndex = strrpos($email, "@");
    if (is_bool($atIndex) && !$atIndex)
    {
        $isValid = false;
    }
    else
    {
        $domain = substr($email, $atIndex+1);
        $local = substr($email, 0, $atIndex);
        $localLen = strlen($local);
        $domainLen = strlen($domain);
        if ($localLen < 1 || $localLen > 64)
        {
            // local part length exceeded
            $isValid = false;
        }
        else if ($domainLen < 1 || $domainLen > 255)
        {
            // domain part length exceeded
            $isValid = false;
        }
        else if ($local[0] == '.' || $local[$localLen-1] == '.')
        {
            // local part starts or ends with '
            $isValid = false;
        }
        else if (preg_match('/\.\./', $local))
        {
            // local part has two consecutive dots
            $isValid = false;
        }
        else if (!preg_match('^[A-Za-z0-9-\-\.]+$', $domain))
        {
            // character not valid in domain part
            $isValid = false;
        }
        else if (preg_match('/\.\./', $domain))
        {
            // domain part has two consecutive dots
            $isValid = false;
        }
        else if (!preg_match('/^[A-Za-z0-9-\-\.]+/\$/', str_replace("\"","\"",$local)))
        {
            // character not valid in local part unless
            // local part is quoted
            if (!preg_match('/^[A-Za-z0-9-\-\.]+/\$/', str_replace("\"","\"",$local)))
            {
                $isValid = false;
            }
        }
        if ($isValid && !(checkdnsrr($domain,"MX") ||
```
```
if (checkdnsrr($domain,"A")) {
    // domain not found in DNS
    $isValid = false;
}
return $isValid;
```

Spread the word! There is some danger that common usage and widespread sloppy coding will establish a de facto standard for e-mail addresses that is more restrictive than the recorded formal standard. If you want to fool the spambots, adopt an e-mail address like, {^c\@**Dog^}@cartoon.com. Unfortunately, you might fool some legitimate e-commerce sites as well. Which do you suppose will adapt more quickly?