Regex 101

This section will talk about some regex tips to get you started. To match *any lowercase* character from the English alphabet, the regex pattern is `[a-z]`. We can deconstruct it as follows:

• The square brackets indicate that you're trying to match *one character* within the set of characters inside of them. For example, if we're trying to match any vowel of the English alphabet, we construct our regex as follows: `[aeiou]`. The order of the characters doesn't matter, and it will match the same.
• Square brackets can also accept a range of characters by adding a hyphen, as seen in our original example.
• You can also mix and match sets of characters within the bracket. `[a-zA-Z]` means you want to match any character from the English alphabet regardless of case, while `[a-z0-9]` means you want to match any lowercase alphanumeric character.

We also need to talk about regex operators. The simplest one is the wildcard operator, denoted by `.` . This means regex will match *any* character, and it's quite powerful when used with the operators `*`, `+` , and `{min,max}`. The asterisk or star operator is used if you don't care if the preceding token matches anything or not, while the plus operator is used if you want to make sure that it matches at least once. The curly braces operator, on the other hand, specifies the number of characters you want to match. Let's look at the following examples:

◇ To match a string that is alphanumeric and case insensitive, our pattern would be `[a-zA-Z0-9]+`. The plus operator means that we want to match a string, and we don't care how long it is, as long as it's composed of letters and numbers regardless of their case.
◇ If we want to ensure that the first part of the string is composed of letters and we want it to match regardless if there are numbers thereafter, it would be `^[a-zA-Z]+[0-9]*$`. The `^` and `$` operators are called anchors, and denote the start and end of the string we want to match, respectively. Since we wanted to ensure that the start of the string is composed of only letters, adding the caret operator is required.
◇ If we want to match just lowercase letters that are in between 3 and 9 characters in length, our pattern would be `^[a-z]{3,9}$`.
◇ If we want a string that starts with 3 letters followed by *any* 3 characters, our pattern would be `^[a-zA-Z]{3}.{3}$`.

There's also the concept of grouping and escaping, denoted by the `()` and the `\` operators, respectively. Grouping is done to manage the matching of specific parts of the regex better while escaping is used so we can match strings that contain regex operators. Finally, there's the `?` operator, which is used to denote that the preceding token is optional. Let's look at the following example:

• If we want to match both *www.tryhackme.com* and *tryhackme.com*, our pattern would be `^(www\.)?tryhackme\.com$`. This pattern would also avoid matching *.tryhackme.com.*
• `^(www\.)?`: The `^` operator marks the start of the string, followed by the grouping of www and the escaped `.`, and immediately followed by the question mark operator. The grouping allowed the question mark operator to work its magic, matching both strings with or without the *www.* at the beginning.
• `tryhackme\.com$`: The `$` operator marks the end of the string, preceded by the string tryhackme, an escaped `.`, and the string com. If we don't escape the `.` operator, the regex engine will think that we want to match any character between tryhackme and com as well.

It's also imperative to note that the wildcard operator can lead to laziness and, consequently misuse. As such, it's always better to use character sets through the brackets especially when we're validating input as we want it to be perfect.

Here's a table to summarize everything above:

[]	**Character Set: matches any single character /range of characte-rs inside**
.	Wildcard: matches any character
*	Star / Asterisk Quantifier: matches the preceding token zero or more times
+	Plus Quantifier: matches the preceding token one or more times
{min,max}	Curly Brace Quantifier: specifies how many times the preceding token can be repeated
()	Grouping: groups a specific part of the regex for better manage-ment

[]	**Character Set: matches any single character /range of characte-rs inside**
\	Escape: escapes the regex operator so it can be matched
?	Optional: specifies that the preceding token is optional
^	Anchor Beginning: specifies that the conseque-nt token is at the beginning of the string
$	Anchor Ending: specifies that the preceding token is at the end of the string

Client Side filtering

using Regex to filter input from client side :

```
<html>
<input type="text" name="name" placeholder="Marcky Marc" required minlength="4" maxlength="70" size="30"><br>  ----------------- 1

<input type="date" name="birthday" required min="1900-01-01" max="2014-12-31"><br>    ----------------------------------    2

<input type="email" name="email" placeholder="marckymarc@tryhackme.com" required pattern=".+@tryhackme\.com" size="30"><br> ------ 3

<input type="number" name="age" placeholder="55" required min="8" max="122"><br>

<input type="url" name="favewebsite" placeholder="https://tryhackme.com" required pattern="https://tryhackme\.com/room/[a-zA-Z0-9]+" size ="50"><br>

<textarea rows="10" cols="20" name="comments" placeholder="Tell us more about yourself"></textarea>
</html>
```

The changes done above incorporate both semantic checking and whitelisting techniques to the existing syntax checking in the prior implementation in order to further filter out allowed values that can be provided in the input fields.

1. The name field, for instance, has been written to allow a maximum of 70 characters - still quite long but somewhat reasonable. This can further be filtered granularly by blacklisting numbers and special symbols, but for our purposes, let's accept this one for now.
2. The date field is tweaked, so the earliest date it would allow is in the year 1900, while the latest date is in 2014, with the working assumption that the youngest user of the website would be eight years old.
3. The email field has also been revamped to only accept tryhackme emails, while the URL field is further narrowed down to only accept tryhackme rooms as the answer to the 'favewebsite' field.

This is the reason why we haven't touched the `<textarea>` field at the very end of the code block. Free-form text is unique in such a way that applying syntax and semantic checks are pretty much impossible. In order to secure this one, we would first need to define what content (e.g. characters, character sets, symbols, etc.) is acceptable and then perform the actual whitelisting. Even then, it wouldn't be enough as it's still a very open field, and so techniques such as output escaping and using parametrized queries, among others, should be implemented as well.

Server Side Filtering using PHP

Our discussion until this point has been geared towards using input validation to ensure that the pieces of data that we're getting from the users are not garbage. Proper input validation on the client-side limits misuse and minimizes the attack surface of the application, however, it doesn't actively cover malicious use cases.

As opposed to the above examples of validating input implemented on the client-side, output escaping and using parametrized queries are implemented on the server-side, adding computational load to the server, but ultimately helping in securing the application as a whole. This is done as an exercise of inherent distrust in the user-provided input despite validation.

The example below uses the built-in `htmlentities()` PHP function to escape any character that can affect the application.

```
$name = htmlentities($_GET['name'], ENT_QUOTES | ENT_HTML5, "UTF-8")
```

On the other hand, the one below uses the HTMLPurifier library to help with some use cases, such as our `<textarea>` conundrum above.

```php
<?php

require_once '/path/to/HTMLPurifier.auto.php';
$config = HTMLPurifier_Config::createDefault();
$purenow = new HTMLPurifier($config);

$output = $purenow->purify($input)

?>
```

Both examples are done on the server-side to escape characters and purify content, respectively both effective ways to prevent user misuse and XSS attacks. Implementing both client and server-side validations ensure that no stones are left unturned.

This exact case highlights the importance of layering defenses and shows point-blank the limitation of client-side input validation. It also gives rise to the notion that input validation is not a one-stop shop for securing your application.

Specific cases warrant specific security requirements, and so on this factor alone, it's already apparent that there exist multiple ways of attacking the challenge of securing user-provided input, and most of the time, they are implemented on top of each other.

----------Checkout Advent of Cyber 2022 Day 17 secure coding.

Not All Solutions are Equal

Input validation is an important layer of security that all production-level applications should have. However, no matter how 'perfect' your input validation is for your specific use case, there simply are limitations to all kinds of security implementations - and input validation is not exempt from it. You cannot make your application fully XSS-proof through input validation alone. Controls may be put in place at the input level that may lessen the attack surface of the application, but it doesn't fully remediate it. Full remediation of an XSS vulnerability in your application requires additional layers of defenses, such as mitigation on the browser-level (via secure cookies, content-security-policy headers, etc.) and escaping and purifying user-provided input.

In light of these remediation techniques, it makes sense to tackle each challenge on its own - SQL and LDAP-related vulnerabilities are addressed differently from XSS, so why try to fit them all in the same formula?