Regex

Regex patterns are used to validate input, such as the exampled used in the portfolio 'post code' script. While Regex patterns can be a powerful tool they can also be exploited by hackers or attackers.

1. **What is ReDOS and what part do 'Evil Regex' play?**

   ReDOS is a regular denial of service. ReDOS attacks slow down the server by presenting malicious strings to be processed by the regex function within the system. A vulnerable regex pattern can also be described as 'Evil Regex'. This is when a developer provides an inefficient regex pattern. Inefficient regex patterns can take a long time to evaluate, often caused by inefficient loops in the code. Therefore, an attacker can supply long, malicious strings for evaluation that cause the system to slow down or even halt, creating a denial-of-service attack.

2. **What are the common problems associated with the use of regex? How can these be mitigated?**

   Developers should aim to avoid repeated loops through regex expressions. There is a specific link to the efficiency of the patterns and the vulnerability of the regex pattern. Efficient patterns avoid examining every character in a string. Similarly, they avoid replicating a search once a rule has been met. The routines should also exit the checking function as soon as one rule has failed. There is direct link to the principals of cyclomatic complexity and the vulnerability of a regex expression.

3. **How and why could regex be used as part of a security solution?**

   Regex expressions can aid security and are excellent for checking the strength of a password. Regex has been used to test the strength of a password in the main coding project. The code is provided in appendix 1. The code includes checks for:

   - 1 number
   - 1 uppercase letter
   - 1 lowercase letter
   - No spaces
   - No non-alphanumeric characters

   This code could be improved further by exiting the routine at a fail point to prevent unnecessary checks and it would be helpful to limit the size of the allowed string (to say 10 characters) which is sufficient to generate up to 8.4×10^{17} combinations.

Appendix 1 – Example of Regex Security

```
def password_check(password):
  # function to check the validity of a new password

  # list of failed checked is set to be empty
  check = []

  # defines the minimum number of characters for an allowed password
  min_length = 3
  # checkd password length
  if len(password) < min_length:
    # adds an error message if the check fails
    check.append('Passwords must contain at least %s characters'
          % (str(min_length)))

  # defined a RegEx pattern contains a digit
  digit_test = r'\d'
  # searches the password for a digit
  if re.search(digit_test, password) is None:
    # adds an error message if the check fails
    check.append('Passwords must contain at least 1 number')

  # defined a RegEx pattern to contain a space
  space_test = r'\s'
  # searches the password for a space
  if re.search(space_test, password) is not None:
    # adds an error message if the check fails
    check.append('Passwords must not contain any spaces')

  # defined a RegEx pattern to contain a capital letter
  caps_test = r'[A-Z]'
```

```
# searches the password for a capital letter
if re.search(caps_test, password) is None:
    # adds an error message if the check fails
    check.append('Passwords must contain at least one captial letter')

# defined a RegEx pattern to contain a lowercase letter
lower_test = r'[a-z]'
# searches the password for a lowercase letter
if re.search(lower_test, password) is None:
    # adds an error message if the check fails
    check.append('Passwords must contain at least one lower case letter')

# defined a RegEx pattern to contain only alpha-numeric charachers
alpha_num_test = r'^[a-zA-Z0-9]+'
# checks the password against the pattern
x = re.match(alpha_num_test, password).group()
if x != password:
    # adds an error message if the check fails
    check.append('Passwords must not cointain any special characters')

# returns a list of errors
return (check)
```