NOTE: SCENARIO IS WHAT THE PUZZLE TAKER SEES.

SCENARIO:

You are developing an account-management service to be used for account registration and authentication. To register an account, a client has to provide a username and a password. According to the registration guidelines, a username can be any non-empty string. The username cannot appear in the password. The `register` method takes two arguments: `String user` (the username of the account), and `String pass` (the password of the account). After checking the validity of the username and password, if both are valid, the account is registered in the database. Otherwise, the method throws an `IllegalArgumentException`. Consider the snippet of code below and answer the following questions, assuming that the code has all required permissions to execute.

```
01   // OMITTED: Import whatever is needed.
02   public final class AccountManager {
03     public static void register (String user, String pass) {
04       if (user == null || user.isEmpty())
05         throw new IllegalArgumentException("Bad Username!");
06       if (pass == null || pass.isEmpty())
07         throw new IllegalArgumentException("Bad Password!");
08
09       // Use regex to check the similarity.
10       Pattern pattern = Pattern.compile("(.*?)" + user);
11       boolean similar = pattern.matcher(pass).matches();
12
13       if (similar)
14         throw new IllegalArgumentException("Weak Password!");
15
16       // OMITTED: Continue to create the account in the database.
17     }
18   }
```

Questions:
1. What will the `register` method do when executed?

2. Which one statement is correct if the `register` method gets called as:
`register("(a+)+", "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!");`

a. The password is checked and the account is created.
b. The method throws an `IllegalArgumentException` with “Bad Password!”
c. The method throws an `IllegalArgumentException` with “Weak Password!”
d. The method throws a `PatternSyntaxException`.
e. None of the above.

[Other statistical questions will be imported here while creating the survey]

NOTE: ANSWER IS TO BE SHOWN TO THE PUZZLE TAKERS AT THE END OF THE SESSION.

ANSWER:
e
The given username (being used as the regex pattern) is a potential evil regex. Having the username, the process of matching (line 11) takes too long to finish and keeps the current thread busy, meaning the call given in the question keeps running (almost) forever. After multiple attempts to call the `register` method with the same arguments, the application might reach the maximum number of running threads and become unresponsive.

NOTE: THE REST OF THIS DOCUMENT CONTAINS EXTRA INFORMATION FOR THE PROJECT RESEARCHERS. IT IS NOT TO BE SHOWN TO THE PUZZLE TAKERS.

TAGS:
java, regex-class, regular-expression, evil-regex, redos, input-verification

CATEGORIES:
Blindspot - YES
Type - Regex
Number of distinct functions - 4
Number of total functions - 5
Blindspot function - `Matcher.matches()`
Blindspot type - Incorrect usage
Number of Parameters in the blindspot function - 0 parameters
Cyclomatic complexity - 5

NAME:
Pattern class, Matcher class - Pattern is a compiled representation of a regular expression. Matcher is an engine that performs match operations.

DESCRIPTION:
A regular expression, specified as a string, must first be compiled into an instance of the `Pattern` class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression.

BLINDSPOT:
While working with regular expression, evil regex patterns (used for ReDoS attack) may cause a blindspot. The Regular expression Denial of Service (ReDoS) is an attack that exploits the fact that most regular expression implementations may reach extreme situations that cause them to work very slowly (exponentially related to input size). By using an evil

regex, an attacker can make a program enter and hang in these extreme situations for a very long time.

CORRECT USE EXAMPLE:

Code:

```
import java.util.*;
public class Application {
  public static void main ( String... args ) {
    Matcher m = Pattern.compile("a*b").matcher("aaaaab");
    boolean b = m.matches();
    // Pattern p = Pattern.compile("(a+)+");
    // Matcher m = p.matcher("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!");
    // boolean b = m.matches(); // It never ends
  }
}
```

MORE INFORMATION:
#N/A

REFERENCES:
1. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS