NOTE: SCENARIO IS WHAT THE PUZZLE TAKER SEES.

SCENARIO:

You receive a file you've been expecting from your friend via email. After downloading the file, to make sure that it has not been tampered with, you calculate the SHA256 hash of the file and compare it with the already known valid hash (which you had received in advance over a secure channel). The following is a simple implementation of an auxiliary method that accepts a file path and an expected hash string, calculates the MD5 hash of the file, compares it to the expected hash, and returns `True` if the two hash values are equal and `False` otherwise. Consider the snippet of code below and answer the following questions, assuming that the code has all required permissions to execute.

```
01   # import whatever is needed
02   BLOCKSIZE = 65536
03
04   def is_valid_file(file_path, expected_hash):
05     hasher = hashlib.sha256()
06     with open(file_path, 'rb') as file:
07       buf = file.read(BLOCKSIZE)
08       while len(buf) > 0:
09         hasher.update(buf)
10         buf = file.read(BLOCKSIZE)
11     return hasher.hexdigest() == expected_hash
```

Questions:

1. What will the function `is_valid_file` do when it is called?

2. Which one of the following is correct if it is possible for you to precisely measure the execution time of the `is_valid_file` function, called with same-sized files and same expected hash value? (Note that there is no external factor affecting the execution time.)

a. The execution time is a constant value.
b. The execution time might vary at each invocation of the method.
c. The execution time increases by the number of calls to the function.
d. The execution time decreases by the number of calls to the function.
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 TAKER AT THE END OF THE SESSION.

ANSWER:
1. The code will calculate the hash of the given file and then compare it to the given expected hash value using the string equality operator.

2. b
In the absence of external factors, even though the size of the files is considered constant, the execution time of the function will not necessarily be the same for each invocation. The reason is that to compare the hash values, the code is using a string equality operator, for which execution time depends on the content of operands. Thus the implementation is vulnerable to timing attacks. In this type of attack, by measuring the execution time you can gradually find the expected hash value (one character at a time). To quote from Python documentation: "When comparing the output of `digest` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest` function instead of the == operator to reduce the vulnerability to timing attacks."

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

TAGS:
Python, cryptography, hash, timing-attack

CATEGORIES:
Blindspot - YES
Type - Crypto
Number of distinct functions - 7
Number of total functions - 7
Blindspot function - `compare_digest`
Function call omitted - YES
Blindspot type - Missing function call
Number of parameters in the blindspot function - 2 parameters
Cyclomatic complexity - 5

NAME:
hmac.compare_digest(a, b)

DESCRIPTION:
Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behavior, which makes it appropriate for cryptography. Both `a` and `b` must be of the same type, either unicode or a bytes-like object.

BLINDSPOT:
The problem here is that generic string comparison functions return as soon as they find a difference between the strings. If the first byte is different, they return after just looking at one byte of the two strings. If the only difference is in the last byte, they process both entire strings before returning. This speeds things up in general, which is normally good.

But it also means someone who can not tell how long it takes to compare the strings can make a good guess where the first difference is.

CORRECT USE EXAMPLE:
Use the `compare_digest` function instead of the == operator to reduce the vulnerability to timing attacks.

MORE INFORMATION:
#N/A

REFERENCES:
1. https://docs.python.org/2/library/hmac.html
2. https://codahale.com/a-lesson-in-timing-attacks/
3. https://security.stackexchange.com/questions/83660