EMA - Test Report

By Thomas Smith

Overview

For simplicity and to avoid issues with integration of my program with a separate testing suite, the test cases are built into the program. This also provides the additional security confidence that the tests are run and results displayed each time the program is executed.

As the program is designed as a Jupyter notebook with individual code cells, the test cases are situated throughout the code so that the test case can be easily run immediately after its associated action.

Due to time restraints, no formal requirements were written against which to create test cases and so all test cases are the result of exploratory testing based on key areas of the program.

Results Overview

TCA01: PASS
TCA02: PASS
TCA03: PASS
TCA04: PASS
TCA05: PASS
TCA06: PASS
TCA07: PASS
TCA08: PASS
TCA09: PASS
TCA10: PASS
TCA11: PASS
TCA12: PASS
TCA13: PASS

Tests Passed: 100%
Tests Failed: 0%
Tests Blocked: 0%

Test Cases

TCA01 - Test Case 1: Confirm that the keys are generated

Code

```
print("TCA01 - Test Case 1: Confirm that the keys are generated -
Results:")
if private_key is not None and public_key is not None:
  print(f"Pass: The keys have been generated. \nThe private key is:
{private_key} \nThe public key: is {public_key}")
else:
  print("Fail: The keys have not been generated.")
```

Output

```
TCA01 - Test Case 1: Confirm that the keys are generated - Results:
Pass: The keys have been generated.
The private key is: PrivateKey(23810... [Omitted for conciseness]
The public key: is PublicKey(23810... [Omitted for conciseness]
```

Result

All tests PASSED

TCA02 - Test Case 2: Confirm that the keys are properly formatted

This test case uses the dictionary qualities of keys generated by Stüvel's RSA library to extract sections of the key that can be (2019). The validation steps are then based on the algorithm as described by Hamza & Kumar (2020) to confirm that the keys are a true pair and are complex enough. To abstract the mathematical functions: greatest common denominator, modular inverse, and prime identification, I used the Sympy library (Meurer, 2017).

Code

```
print("TCA02 - Test Case 2: Confirm that the keys are properly formatted -
Results:")
```

```
n, e = public_key.n, public_key.e # n is the modulus and e is the exponent
d, p, q = private_key.d, private_key.p, private_key.q # d is the modular
inverse of e, and p and q are the prime factors of n

# checks that the keys are a valid pair
if gcd(e, (p-1)*(q-1)) == 1 and (e * d) % ((p-1)*(q-1)) == 1:
  print("Pass: the public and private keys are consistent.")
else:
  print("Fail: the public and private keys do no match.")

# checks that the modulus is long enough to be secure
if n.bit_length() >= 2048:
  print("Pass: the modulus is sufficiently large.")
else:
  print("Fail: the modulus size is not sufficiently large.")

# checks that the factors cannot be broken down into smaller places as
this would make the encryption easier to crack
if isprime(p) and isprime(q):
  print("Pass: the factors of the modulus are prime.")
else:
  print("Fail: the factors of the modulus are not prime.")

# checks that the private key will be able to decrypt the encryption
if d == mod_inverse(e, (p-1)*(q-1)):
  print("Pass: the private exponent is consistent with the modulus.")
else:
  print("Fail: the private exponent is not consistent with the modulus.")
```

Output

```
TCA02 - Test Case 2: Confirm that the keys are properly formatted -
Results:
Pass: the public and private keys are consistent.
Pass: the modulus is sufficiently large.
Pass: the factors of the modulus are prime.
Pass: the private exponent is consistent with the modulus.
```

Results

All tests PASSED

TCA03 - Test Case 3: Confirm that the files have been generated, are the correct file format, and contain the keys

This test case uses the imported library os.path to determine whether files were created

Code

```
# TCA03 - Test Case 3: Confirm that the files have been generated, are the correct file format, and contain the keys
print("TCA03 - Test Case 3: Confirm that the files have been generated, are the correct file format, and contain the keys - Results")

for key in ["public.pem", "private.pem"]:
  if os.path.isfile(key):
    print(f"Pass: {key} file exists.")
  else:
    print(f"Fail: {key} file does not exist.")

with open("public.pem", "rb") as f:
  if public_key == rsa.PublicKey.load_pkcs1(f.read()):
    print(f"Pass: public.pem contains the public key.")
  else:
    print(f"Fail: public.pem does not match the public key.")

with open("private.pem", "rb") as f:
  if private_key == rsa.PrivateKey.load_pkcs1(f.read()):
    print(f"Pass: private.pem contains the private key.")
  else:
    print(f"Fail: private.pem does not match the private key.")
```

Output

```
TCA03 - Test Case 3: Confirm that the files have been generated, are the
correct file format, and contain the keys - Results
Pass: public.pem file exists.
Pass: private.pem file exists.
Pass: public.pem contains the public key.
```

```
Pass: private.pem contains the private key.
```

Results

All tests PASSED

TCA04 - Test Case 4: Confirm that the encrypted message is not the same as the message

Code

```
print("TCA04 - Test Case 4: Confirm that the encrypted message is not the
same as the message - Results:")

if message == encrypted_message:
  print("Fail: message and encrypted message are the same.")
else:
  print(f"Pass: message has been changed. \nThe encrypted message is
{encrypted_message}")
```

Output

```
TCA04 - Test Case 4: Confirm that the encrypted message is not the same as
the message - Results:
Pass: message has been changed.
The encrypted message is b'7H\xf0...[omitted for conciseness]
```

Results

All tests PASSED

TCA05 - Test Case 5: Confirm that the file has been generated, is the correct format and contains the encrypted message

This test case uses the imported library os.path to determine whether files were created.

Code

```
print("TCA05 - Test Case 5: Confirm that the file has been generated, is 
the correct format and contains the encrypted message - Results:")
if os.path.isfile("encrypted.message"):
  print("Pass: encrypted.message exists.")
else:
  print("Fail: encrypted.message does not exist.")

if encrypted_message == open("encrypted.message", "rb").read():
  print("Pass: encrypted.message contains the encrypted message")
else:
  print("Fail: encrypted.message does not contain the encrypted message")
```

Output

```
TCA05 - Test Case 5: Confirm that the file has been generated, is the
correct format and contains the encrypted message - Results:
Pass: encrypted.message exists.
Pass: encrypted.message contains the encrypted message
```

Results

All tests PASSED

TCA06 - Test Case 6: Confirm that the message has been decrypted back to original

Code

```
print("TCA06 - Test Case 6: Confirm that the message has been decrypted 
back to original - Results:")

if decrypted_message.decode() == message:
  print(f"Pass: message has been successfully decrypted. \nOriginal 
message: {message}\nDecrypted message: {decrypted_message.decode()}")
else:
  print(f"Fail: message decrypted unsuccessfully. \nOriginal message: 
{message}\nDecrypted message: {decrypted_message}")
```

Output

```
TCA06 - Test Case 6: Confirm that the message has been decrypted back to
original - Results:
Pass: message has been successfully decrypted.
Original message: this is the message
Decrypted message: this is the message
```

Results

All tests PASSED

TCA07 - Test Case 7: Confirm that the message cannot be decrypted with another key

This is a false-test that checks for a failure. It would usually cause a run-time error to appear and so I decided to use the library ‘pytest’ to catch these errors and present a test pass instead (Krekel et al., 2004).

Code

```
print("TCA07 - Test Case 7: Confirm that the message cannot be decrypted
with another key - Results:")

# Generates another public/private key pair for testing.
public_key_wrong, private_key_wrong = rsa.newkeys(1024)

# Uses the wrong key to attempt to decrypt the message.
with pytest.raises(Exception, match='Decryption failed'):
  rsa.decrypt(encrypted_message, private_key_wrong)

print("Pass: the wrong private key has not decrypted the message")
```

Output

```
TCA07 - Test Case 7: Confirm that the message cannot be decrypted with
another key - Results:
Pass: the wrong private key has not decrypted the message
```

Results

All tests PASSED

TCA08 - Test Case 8: Confirm that signature has been generated

Code

```
print("TCA08 - Test Case 8: Confirm that signature has been generated - Results:")

if message_signed is not None:
  print(f"Pass: signature has been generated. \nThe signature is: {message_signed}")
else:
  print("Fail: signature has not been generated.")
```

Output

```
TCA08 - Test Case 8: Confirm that signature has been generated - Results:
Pass: signature has been generated.
The signature is: b'"\xce\x97...[omitted for conciseness]
```

Results

All tests PASSED

TCA09 - Test Case 9: Confirm that the file has been generated, and contains the signed message

This test case uses the imported library os.path to determine whether files were created

Code

```
print("TCA09 - Test Case 9: Confirm that the file has been generated, and contains the signed message - Results:")
if os.path.isfile("signature"):
  print("Pass: signature file exists.")
else:
  print("Fail: signature file does not exist.")

if message_signed == open("signature", "rb").read():
  print("Pass: signature file contains the signature.")
else:
  print("Fail: signature file does not contain the signature.")
```

Output

```
TCA09 - Test Case 9: Confirm that the file has been generated, and
contains the signed message - Results:
Pass: signature file exists.
Pass: signature file contains the signature.
```

Results

All tests PASSED

TCA10 - Test Case 10: Confirm that the public key can verify the message using the signature

This is a false-test that checks for a failure. It would usually cause a run-time error to appear and so I decided to use the library ‘pytest’ to catch these errors and present a test pass instead (Krekel et al., 2004).

Code

```
# TCA10 - Test Case 10: Confirm that the public key can verify the message
using the signature
print("TCA10 - Test Case 10: Confirm that the public key can verify the
message using the signature - Results:")

if rsa.verify(message_unsigned.encode(), message_signed, public_key) ==
"SHA-256":
  print("Pass: the message was verified using the signature and public
key.")
else:
  print("Fail: the message could not be verified using the signature and
public key.")
```

Output

```
TCA10 - Test Case 10: Confirm that the public key can verify the message
using the signature - Results:
Pass: the message was verified using the signature and public key.
```

Results

All tests PASSED

TCA11 - Test Case 11: Confirm that the wrong message cannot be verified

This is a false-test that checks for a failure. It would usually cause a run-time error to appear and so I decided to use the library ‘pytest’ to catch these errors and present a test pass instead (Krekel et al., 2004).

Code

```
print("TCA11 - Test Case 11: Confirm that the wrong message cannot be verified - Results:")

message_wrong = "This is the fake message"

# catches the error and presents a pass for testing cleanliness
with pytest.raises(Exception, match='Verification failed'):
  rsa.verify(message_wrong.encode(), message_signed, public_key)

print("Pass: the wrong message could not be verified using the signature and public key.")
```

Output

```
TCA11 - Test Case 11: Confirm that the wrong message cannot be verified -
Results:
Pass: the wrong message could not be verified using the signature and
public key.
```

Results

All tests PASSED

TCA12 - Test Case 12: Confirm that the wrong public key does not verify the message

This is a false-test that checks for a failure. It would usually cause a run-time error to appear and so I decided to use the library ‘pytest’ to catch these errors and present a test pass instead (Krekel et al., 2004).

Code

```
print("TCA12 - Test Case 12: Confirm that the wrong public key does not verify the message - Results:")

# catches the error and presents a pass for testing cleanliness
with pytest.raises(Exception, match='Verification failed'):
  rsa.verify(message_unsigned.encode(), message_signed, public_key_wrong)

print("Pass: the message could not be verified using the right signature but wrong public key.")
```

Output

```
TCA12 - Test Case 12: Confirm that the wrong public key does not verify the message - Results:
Pass: the message could not be verified using the right signature but wrong public key.
```

Results

All tests PASSED

TCA13 - Test Case 13: Confirm that the wrong signature does not verify the message

This is a false-test that checks for a failure. It would usually cause a run-time error to appear and so I decided to use the library ‘pytest’ to catch these errors and present a test pass instead (Krekel et al., 2004).

Code

```
print("TCA13 - Test Case 13: Confirm that the wrong signature does not verify the message - Results:")
```

```python
message_signed_wrong = rsa.sign(message_unsigned.encode(),
private_key_wrong, "SHA-256")

# catches the error and presents a pass for testing cleanliness
with pytest.raises(Exception, match='Verification failed'):
  rsa.verify(message_unsigned.encode(), message_signed_wrong, public_key)

print("Pass: the wrong message could not be verified using the right
public key but wrong signature.")
```

Output

```
TCA13 - Test Case 13: Confirm that the wrong signature does not verify the
message - Results:
Pass: the wrong message could not be verified using the right public key
but wrong signature.
```

Results

All tests PASSED

References

Adeniyi, E. A., Falola, P. B., Maashi, M. S., Aljebreen, M. & Bharany, S. (2022) Secure sensitive data sharing using RSA and ElGamal cryptographic algorithms with hash functions. *Information* 13(10): 442.

Anderson, R. (2020) *Security Engineering: A Guide to Building Dependable Distributed Systems*. 3rd ed. Indianapolis: Wiley Publishing.

Boudot, F., Gaudry, F., Guillevic, A., Heninger, N., Thomé, E., Zimmermann, P., et al. (2020) Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. *Advances in Cryptology – CRYPTO 2020*: 62-91.

Brookshear, J. G. & Brylow, D. (2020) *Computer Science: an overview*. 13th ed. New York: Addison Wesley Longman Inc.

Hamza, A. & Kumar, B. (2020) A Review Paper on DES, AES, RSA Encryption Standards. *9th International Conference System Modeling and Advancement in Research Trends (SMART)*: 333-338

Huang, L. S., Adhikarla, S., Boneh, D. & Jackson, C. (2014) An experimental study of TLS forward secrecy deployments. *IEEE Internet Computing* 18(6): 43-51.

Klíma, V., Pokorný, O. & Rosa, T. (2003) Attacking RSA-based sessions in SSL/TLS. *International Workshop on Cryptographic Hardware and Embedded Systems*: 426-440.

Krekel, H., et al. (2004) How to write and report assertions in tests. Available from https://docs.pytest.org/en/7.1.x/how-to/assert.html#assertions-about-expected-exceptions [Accessed on 10 February 2024].

Leurant, G. & Peyrin, T. (2020) SHA-1 is a Shambles: First Chosen-prefix Collision and Application to the PGP Web of Trust. *29th USENIX security symposium (USENIX security 20):* 1839-1856

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S.B. & Rocklin, M. (2017) SymPy: symbolic computing in Python. *PeerJ Computer Science.*

Stüvel, S. A. (2019) Python-RSA 4.8 documentation. Available from https://stuvel.eu/python-rsa-doc/index.html [Accessed on 5 February 2024].