Part 1 Tests Feedback

Common Mistakes

Here is a selection of common mistakes made in Part 1.

Attempting to use asymmetric private key as symmetric key. The asymmetric private key is an RSA key object, which is a tuple of integers (n, e, d, p, q, u) (the modulus, public exponent, private exponent, factors of n, and the CRT coefficient $(1/p) \bmod q$)—it is only useful for the asymmetric methods. A symmetric key is just a random string of bytes. These should be generated using `get_random_bytes()`.

A related mistake was to generate symmetric keys by converting the private key to a string and hashing it. While this isn't necessarily insecure, it is a bit brittle—your symmetric keys are inherently tied to your private key, making them impossible to change. For this reason, cryptographers would probably consider this approach to be bad practice.

Keeping extra state in the client. Per the project spec, you aren't allowed to keep extra state in the client except as an optimization (such as a locally cached copy of state stored on the server). In the provided functionality tests, `t2` tested whether two instances of the client could download a file uploaded by the other, assuming both instances have the same username and RSA private key. Be careful to make all state recoverable from the server.

For instance, the idea is that Alice should be able to install a copy of the client on her work machine and on her home machine, and as long as she securely transfers her private key to both instances, everything should just work: both instances of the client should have access to all her files.

Length restrictions on asymmetric encryption. Directly encrypting the filename or file contents using asymmetric RSA encryption won't work. Both the filename and file contents can be arbitrarily long. However, RSA-OAEP (and many other asymmetric encryption schemes) are length-limited: they can't handle arbitrarily-long inputs. For the provided `asymmetric_encrypt()` function, the upper limit is 190 bytes.

Using hash(name) for the ids of files. Because filenames may be predictable (low entropy), using hash(name) can leak the contents of the name, violating the confidential-

ity requirement. In particular, hash(name) allows dictionary attacks on the filename: an adversary who can enumerate a set of candidate filenames can determine which of those candidates is correct. Two possible fixes are described in the sample designs: ids should instead of chosen randomly (Design 1), or be the HMAC of the name (Design 2).

Using encryption without authentication. Encrypting file contents, without also providing authentication to detect tampering, is insecure. It allows sophisticated chosen-ciphertext attacks. These attacks can potentially even allow an attacker to learn the contents of the file. If you're interested to learn more, you can read about padding oracle attacks.

A related flaw is to use a MAC, but fail to check it, or to decrypt before verifying the MAC. For instance, consider an implementation of download() that first decrypts the encrypted file, then checks the MAC and raises IntegrityError if the MAC tag is invalid. This might sound safe, but it is actually quite dangerous. A malicious storage server could tamper with the ciphertext, then ask the client to download the file. An error will happen at some point, but if the adversary can distinguish whether the error happened during decryption (e.g., CryptoError due to invalid padding) vs. during MAC verification (e.g., IntegrityError thrown by your client), then padding oracle attacks may again become possible. To defend against this: compute a MAC on the ciphertext, and verify the validity of the MAC *before* decrypting the ciphertext.

Using hashes for integrity instead of MACs. Hashes are not a substitute for MACs. A problem with hashes is that they can be recomputed by a malicious storage server. The server can modify the data and then also modify the hash to match, making the modifications go undetected by the client. On a related note, it is difficult to design your own hash-based MAC. There are many attacks on schemes that aren't quite HMAC. You should use a provided MAC function instead of designing your own.

Generating symmetric keys as a hash of the filename. Some generated their symmetric keys for encrypting the file using $H(\text{username}||\text{filename})$. This suffers from the same problems as using $H(\text{filename})$ for the id. Both usernames and filenames are low entropy and easy to guess, making the resulting symmetric keys easy to guess. Symmetric keys should be randomly generated using `get_random_bytes()`.

Using CTR mode but not setting an IV in the counter. CTR mode is insecure if you ever reuse a nonce, over all encryptions under the same key. The easiest way to set an IV is to generate random bytes and pass them as the prefix to `new_counter`. It is also possible to use the IV to set the initial value of the counter. You can prepend this IV to the ciphertext like you would in CBC mode encryption, and then use it to re-create the counter object during decryption. CTR mode without an IV is insecure; it is vulnerable to the same attacks as if you were to re-use the pad in a one-time pad scheme (the "two-time pad").

Using insecure primitives. DES and RC4 encryption are broken. Encrypted messages stored with these algorithms can be decrypted without the key. MD2, MD5, and SHA-1 are cryptographically broken hash functions and should not be used (and they shouldn't be used for HMAC-based schemes, either: though no attack is known that can forge MAC tags for these schemes, their use is discouraged). Use of these primitives is detected by tests **BrokenTestBadCrypto** and **WeakTestBadCrypto** (see below). Also, ECB mode should be avoided for file encryption—it leaks information about the file contents. Modes like CBC and CTR (which we've covered in class) provide full IND-CPA security if used correctly.

Storing data in plaintext. Storing the file contents or filename on the storage server in plaintext (in unencrypted form) is not secure. If it is sent to the storage server, the storage server can see it. Using the filename as the id for get()/put() reveals the filename. Similarly, cryptographic keys should not be stored on the storage server unencrypted.

Test Case Explanations

Our autograder included numerous security tests. Here is an explanation of what each one is testing and some sample design flaws that might be responsible for failing that test case.

CheckDifferentialAttacks. Check if there are any values stored on the server that leak any bits of the message. We do this using a technique similar to differential cryptanalysis, which will catch any submissions that store the plaintext in a non-standard manner (e.g., compressed, pickled, ROT13d, etc).

CheckEncryptionMACDifferentKey. MAC keys and encryption keys should be different, and this test case checks just that.

CheckKeyNotStoredPlaintext. The point of creating an encryption key is to hide the content of a message. This test verifies that the encryption keys are never stored, in plaintext, on the server.

CheckPaddingOracle. A padding oracle attack is a chosen-ciphertext attack: if the attacker can convince the client to decrypt (tampered) ciphertexts chosen by the attacker, then a padding oracle attack allows the attacker to decrypt an encrypted message. In this attack, the attacker modifies the ciphertext in some cleverly chosen fashion, asks the client to decrypt it, and then observes the decryption process caused an invalid-padding error. If the attacker can observe whether such an error occurred, then this leaks partial information; after repeating this many times, an attacker can piece together all of these clues to deduce what the original message must have been. These attacks are very powerful and are subtle if you don't know about them.

Failure of this test case typically indicates that you used encryption without authentication, or that you decrypted some ciphertext without checking the MAC or before verifying the validity of the MAC. Defend against padding oracle attacks by always verifying the MAC *before* decrypting message contents.

CheckPlainttextMAC. MACs provide integrity, but they don't promise to provide confidentiality for their input. Therefore, you should generally avoid including plaintext data directly as part of the input to the MAC, if the MAC is not encrypted, as this might leak plaintext bits. Also, computing a MAC solely on the file contents loses IND-CPA security: the same file contents will always yield the same MAC.

CreateFakeKey. It is important to bind the name of a file to its contents. Otherwise, if a user creates files A and B, a malicious server could swap the names and have downloads for A return the content of B and vice versa.

RandomSwapValues. This test randomly permutes the ids and values stored on the storage server. For instance, if you have stored {a:b, c:d} on the server, this test might change it to {a:d, c:b}. Then, it will try downloading files. If download() returns incorrect data, then the test case fails. Failing this test case tends to indicate that you are missing a MAC, or you have not bound the MAC'ed value to the filename it is associated with. The fix is generally to include the filename or id in the input to the MAC, as we did in Design 1 and Design 2.

EncryptionHasNoncesBlackBox. Verifies that encrypted messages are correctly randomized, using IVs or nonces.

CBCIVsNotReused. Verifies that you don't reuse the same IV more than once for CBC mode encryption per key. Such reuse is insecure and can leak partial information about the plaintexts: the same contents will encrypt to the same ciphertext, which violates IND-CPA security, and if two plaintexts start with the same 16 bytes, then an adversary can detect this (by noticing that the ciphertexts start with the same 16 bytes). Use a fresh, random IV for each CBC encryption you do. Generate the IV using `get_random_bytes()`.

KeysAreChosenFresh. Keys should be chosen randomly using `get_random_bytes()` and should not be generated via any other means. In particular, hashing the private key is poor practice and makes rotating keys difficult if the private key were ever to be compromised.

NameIsNotFromECB. Ids should not be generated by encrypting the filename with ECB mode or CBC/CTR mode with constant IV, as this leaks partial information. In particular, if two filenames start with the same 16 bytes, then the adversary can detect this

(by noticing that the ciphertexts start with the same 16 bytes). Also, these modes allow dictionary attacks: an adversary who can convince you to upload files under a name of the attacker's choice will be able to exploit the properties of ECB/CBC/CTR to figure out the filename of your secret file, if the attacker can identify a list of candidate names for the secret file. Finally, CTR mode with a constant IV is especially bad: it is vulnerable to the attacks on pad reuse with the one-time pad.

NameIsNotFromHash. Verifies that ids are not generated by hashing the filename. That has the same problems outlined above.

SmallCTRUsed. When using CTR mode, the counter should be at least 64 bits, to prevent repetition of the counter value.

CTRUsesUniqueValues. When using CTR mode, the counter values used should be unique. Give either a unique prefix, suffix, or initial value.

StringNotPlaintext. The filename should not directly appear in plaintext on the server. Similarly, the contents of the file should not appear in plaintext on the server.

StringNotPlaintextBoth. You fail this test if both the name and the contents of the file appear in plaintext on the server.

CanEncryptionBeOracle. This test checks that you are not doing anything funky with ciphertexts. We don't take off any points for it, but if you have passed all the functionality tests, it is a warning to double-check your design as the approach seems a bit suspicious. The test checks that it is possible to replace the specifics of `symmetric_encrypt()` with a function that returns random bytes. If your test fails, it means either (a) you fail one of the functionality tests, or (b) when we replace `symmetric_encrypt()` in this nature, you fail a functionality test. If you pass all the functionality tests but fail this test, review your design to see what you are doing with the crypto. If you failed some of the functionality tests, you can ignore this test.

BrokenTestBadCrypto and WeakTestBadCrypto. You fail this test if you use insecure cryptographic algorithms. DES and RC4 are broken; you should not use those algorithms. Encrypted messages stored with these algorithms can be decrypted without the key. MD2, MD5, and SHA are cryptographically weak hash functions and should not be used. They shouldn't be used for HMAC-based schemes, either: though no attack is known that can forge MAC tags for these schemes, their use to build a MAC is discouraged. Instead, use strong algorithms like AES, SHA256, and SHA256-HMAC.

NotIntegrityError. Something not an `IntegrityError` was raised during testing. Tests change data to unexpected values and your code must handle those values by the spec.